View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Louisville

University of Louisville

ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

5-2010

Application of temporal difference learning to the game of Snake.

Christopher Lockhart
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

 Part of the Computer Engineering Commons

Recommended Citation

Lockhart, Christopher, "Application of temporal difference learning to the game of Snake." (2010). *Electronic Theses and Dissertations.* Paper 848.
https://doi.org/10.18297/etd/848

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

APPLICATION OF TEMPORAL DIFFERENCE LEARNING TO THE GAME OF SNAKE

By

Christopher Lockhart
B.S., University of Louisville, 2009

A Thesis
Submitted to the Faculty of the
Graduate School of the University of Louisville
in Partial Fulfillment of the Requirements
for the Degree of

Master of Engineering

Department of Computer Engineering and Computer Science
University of Louisville
Louisville, Kentucky

May 2010

APPLICATION OF TEMPORAL DIFFERENCE LEARNING TO THE GAME OF SNAKE

By

Christopher Lockhart
B.S., University of Louisville, 2009

A Thesis Approved On

Date

by the following Thesis Committee:

Thesis Director

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Richard Sutton and Dr. Andrew Barto for their work in the area of reinforcement learning, this thesis would not have been possible without their intensive research and documented discoveries. This thesis would not be possible if it wasn't for the support and patience I have received from the Computer Science and Computer Department faculty, my friends, and family. I would like to specifically thank my father, Jerry Lockhart. His prolonged encouragement and participation in my life's learning activities has allowed me to reach my highest potential.

ABSTRACT

APPLICATION OF TEMPORAL DIFFERENCE LEARNING TO THE GAME OF SNAKE

Christopher Lockhart

May 2, 2010

The game of Snake has been selected to provide a unique application of the TD(λ) algorithm as proposed by Sutton. A reinforcement learning technique for producing computer controlled players is documented. Using value function approximation with multilayer artificial neural networks and the actor-critic architecture, computer players capable of playing the game of Snake can be created. The adaptation to the standard neural network backpropagation procedure will be documented. Not only does the proposed technique provide reasonable player performance, its application is unique; this approach to Snake has never been documented. By performing sets of trials, the performance of the players are evaluated and compared against an existing machine learning technique. Learning curves provide visualization for the results. Though the snake players are shown to be capable of achieving lower scores than with the existing method, the technique is able to produce agents that accumulate scores, much more efficiently.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

CHAPTER I

INTRODUCTION

A Computers Learning to Play Games

Since the inception of digital games, game developers have been using artificial intelligence (AI) techniques to create computer controlled opponents (also called *game agents*) that enhance a player's experience by providing realistic and human-like competition. In modern times, most digital games have computer controlled elements that use artificial intelligence techniques. While some of the techniques have been widely and successfully used, one area of machine learning that has not gained acceptance in the industry of game development is Reinforcement Learning.

This area of machine learning attempts to capture one of the aspects of the way intelligent biological systems learn: through success and failure (or more technically, exploration and exploitation.) As humans, we experience this process when learning new skills. When we develop skills, we use feedback from our environment to adjust our actions to improve our performance.

It would seem that using Reinforcement Learning would be an appropriate way to develop artificial intelligence in games. A computer capable of learning in-game could enhance the capabilities of game agents. It is easy to imagine a computer opponent, adapting and developing new strategies. Such an opponent would provide more difficulty for human players seeking challenges and provide a more engaging experience.

Today, such a view is not a widespread reality. While some commercial

games exist that implement machine learning algorithms, Galway et al. discusses the number of issues that prevent developers from incorporating machine learning techniques in games. The vast state space and high dimensionality needed to represent today's game environments is a challenge to the effectiveness of current algorithms. In addition, learning is intrinsically unpredictable; once a finished game has been released to the public, the creators do not have full control over which direction the agent will evolve [4]. The uncertainty of the quality of artificial intelligence introduced by machine learning is enough for game developers to pursue more established alternatives.

These issues associated with machine learning primarily keep techniques like Reinforcement Learning in the realm of academia and must be overcome before these techniques become more widely used in the game industry. The academic exploration and evaluation of machine learning techniques will further add value to the knowledge of these techniques.

B Point of Study

In this study, one area of reinforcement learning will be further explored by studying a novel application of temporal difference learning to the game of Snake. Using a neural network approach to the TD(λ) algorithm, as proposed by Sutton, and the actor-critic architecture as described by Barto, a computer agent capable of learning to play the game of Snake will be developed. Through several trials, the performance of the agent and the learning algorithm will be evaluated. Factors will be identified that affect performance and the optimal configuration will be presented and compared to an existing solution.

C Researching Reinforcement Learning with Snake

This section enumerates several reasons why Snake and temporal difference methods were chosen for developing a computer agent capable of intelligent play.

1 Solution Scarcity

There are few sources that document machine learning implemented for the game of Snake. The evaluation of the proposed temporal difference learning technique would increase the number of documented solutions to the game.

The scarcity of solutions is most likely due to the fact that simple programmed policies perform well and are more practical to implement; the snake can achieve a perfect score very easily by following a simple pattern that is easily conceived and implemented [2]. The issue with this approach is that the agent may not appear to simulate human behavior, a quality that is desired in developing computer opponents.

Since artificial intelligence for Snake can be created by less complex methods, the use of reinforcement techniques in this game will likely remain an academic exercise. However, once research has found ways to defeat the difficulty of solving simple problems like these, machine learning techniques may begin to be used in more complicated settings.

2 Unique Application of TD Methods

Implementing the TD(λ) learning algorithm in the game of Snake would produce an application of temporal difference methods that has never been previously documented. If successful, this application could add credibility to using TD methods in games and potentially expand the realm of applications. Regardless of the result, the study will have added more insight to the capabilities and performance of these methods to similar kinds of problems.

In addition, the proposed implementation utilizes multi-layer neural networks for function approximation. Several adaptations and formulations of algorithms will be required. The text will explicitly document changes to existing algorithms such that the reader, interested in implementing the system, will be able to do so.

3 General Expansion of Academic Knowledge

In addition to expanding the solution to the game of Snake, it was felt that an application of a machine learning solution to this game would add to the existing knowledge of the artificial intelligence community. Evaluating and publishing the performance of an Artificial Intelligence technique as applied to a new problem will serve to further contribute to the known abilities of the learning technique. The knowledge obtained in this study can better determine whether this kind of solution is appropriate for future problems.

D Literature Review

1 Useful Cases of Reinforcement Learning in Games

Though there are significant problems with the implementation of machine learning algorithms in games, there are still many documented cases of their use. Galway et al. provides a detailed survey of the use of machine learning in modern games. A published case in 1995 can be considered one of the most successful reinforcement learning implementations. TD-Gammon which used Temporal Difference learning, was capable of playing Backgammon almost as well as world champion players after playing thousands of games against itself.

Now, the focus has shifted. Primarily in the last ten years, the scene of academic research has moved from traditional board games, like Backgammon and Chess, into the realm of modern digital games where their complexity provides new challenges [4].

In the fighting game *Tao Feng: Fist of the Lotus* researchers successfully generated an optimal control policy. By selecting a reward policy that encouraged more aggressive behavior, the computer agent eventually learned to exploit loopholes in deterministic opponents. The selection between exploitation and exploration (See Section B in Chapter II) was critical to the learning performance

of the algorithm [4]. This result will contribute to further consideration of the reward policy selected for use in this study.

In another case, a reinforcement learning strategy that attempted to control RC cars, found that using multilayer neural networks for linear and value function approximations failed to generate an acceptable control policy in all experiments performed [4].

This study may find similar results in determine a suitable control policy for the Snake agent; it is reasonable to expect the task of driving a car is similar to driving the player in Snake. However, there are factors that differentiate between the two tasks: the RC car controller attempts to produce a continuous control output whereas the snake needs only to select three actions per state.

2 Machine Learning in the Game of Snake and Similar Games

There are not a lot of resources specific to this game; however, there are a few which can point out useful considerations for design decisions.

Genetic Programming Solution

Ehlis proposes a unique solution to Snake by using a genetic programming technique. He shows that he can produce an optimal strategy by implementing a genetic algorithm to produce sequences of programming instructions that evolve into well performing solutions.

In constructing a computer agent to learn to play the game of Snake, there are many considerations that can be made by studying Ehlis' technique.

1. The solution elicits the use of a distinct set of features to sample the game state [2]. Initially, Ehlis was only able to produce policies that performed sub-optimally due to the features in the feature set. By adding more descriptive features, the genetic programming algorithm was able to produce a Snake policy capable of achieving a perfect score.

Since the policy was able to achieve an optimal score, it can be inferred that the added set of features were very important to the success of the evolved policies. These features should be considered when constructing a feature based game state representation.

2. The solution uses the concept of *primed runs* to greatly enhance the learning performance of the algorithm by four fold [2]. This involves selecting well performing pre-produced solutions to be placed in the initial breeding pool along with those that have been randomized; this results in stagnant solutions being able to inherit successful strategies.

   A similar strategy for training with neural networks can be adapted. Once a network has been trained to a suitable performance, it can be used as a seed for future training sessions. This will allow further training sessions to progress more quickly to better solutions.

3. In the world of digital games, there is a desire to create computer agents that are capable of displaying human-realistic artificial intelligence. Though the policy created by the genetic solution was optimal, its behavior does not resemble human-competitive results. This could detract from the implementation of the policies derived by the genetic programming solution in a game.

   In the analysis of the reinforcement learning strategy proposed by this paper, observations will be made concerning the human realistic behavior exhibited by the developed agents as a goal for Artificial Intelligence development.

Coevolution and Tron

Funes also used the evolutionary approach to a similar game called Tron. However, his study not only focused on evolving solutions through self-play but *coevolving* them through human interaction as well. The two games are similar

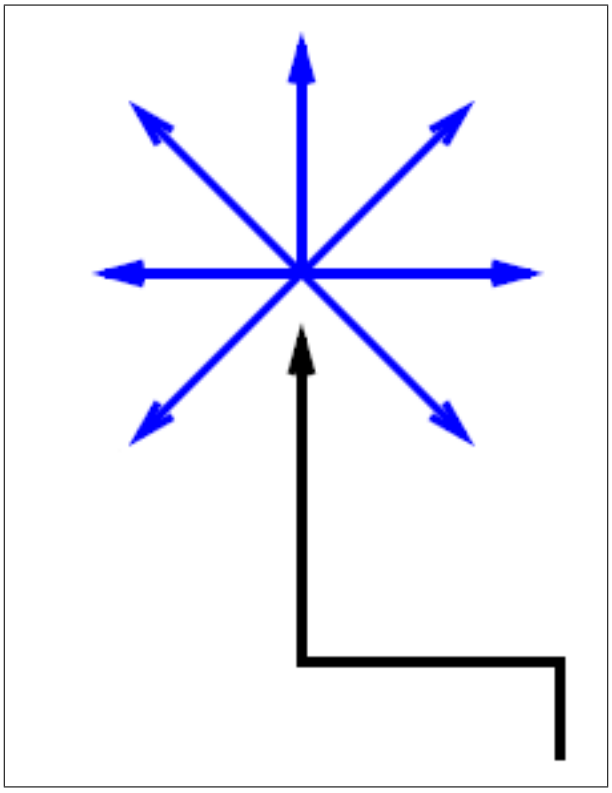


Figure 1. The directions in which a Tron agent can sense its environment. [3]

because the Tron control problem is almost identical to Snake; the player must avoid itself and other players. In Tron, two players try to win the game by surviving the longest.

In the study, Funes proposed a sensory system that allowed the player agents to sample the game's environment. The player was able to sense the distance to the nearest obstacle in seven cardinal directions from the player's current position (See Figure 1.) This is not unlike the feature sampling demonstrated in the previous genetic programming approach: obstacles, at specified distances, could be queried in four cardinal directions around the snake's position.

The results show that the agents were capable of evolving human-like behavior. Although exhibiting similar properties, the agents were also capable of developing behaviors unique to the computer players as well [3].

Even though the agents developed advanced capabilities using this coevolutionary approach (with some achieving the probability of winning greater than 70%) the best performers were still not capable of exhibiting the performance of the best human play. The ability of the agents to learn behaviors that approach human play is commendable and as mentioned, can be desirable.

3 Considerations of Previous Research

Some considerations can be made from the results in the two previous studies. Each study was able to achieve successful results. One was able to produce a computer agent that plays optimally, while the other was able to produce players that approached human behavior with respectable performance. In each study, a simple feature based game state representation was utilized for sampling the game's environment. The reinforcement learning technique as proposed by this paper shall not only evaluate the method with similar representations but also attempt to evaluate the performance of the learning algorithms when the state is more literal. There is ample research that suggests that trying to utilize other than the simplest and most descriptive features in state representation may result in tarnished performance [7]. As mentioned, the tradeoff between exploitation and exploration will be critically examined for this application to the game of Snake.

E Thesis Organization

A brief introduction to the material required for the implementation of the proposed learning system will be presented in the Chapter II along with a proposal for solving Snake with actor-critic architecture with multilayer neural networks. The design and implementation of the proposed technique will be presented in Chapter III. An experiment used to modify factors that are suspected of affecting learning performance will be discussed and results presented in Chapter IV. A comparison and analysis of the system to an existing machine learning technique is

covered in Chapter V. Conclusions, and recommendations will be presented in Chapter VI.

CHAPTER II

BACKGROUND MATERIAL

This chapter will provide the necessary theoretical foundation for the implementation of the learning system.

A A Description of the Game

The game of Snake was popularized by its appearance on Nokia cell phones in 1997 but has been recreated in several forms for multiple computing platforms since 1978 [2]. The game is very simple; the snake is controlled by the player and is allowed to move inside a 2-dimensional playing field surrounded by walls. At each discrete interval, called a time step or step, the snake must move forward, turn left, or turn right as the game requires that the snake cannot stop moving. The game will randomly generate and place one piece of food in the game field at a time. When the snake moves onto a piece of food, it is eaten and the snake's length grows by one.

The goal is to eat as many pieces of food without ending the game by colliding the snake into itself or the walls. The consumption of food increases the score by one for every piece eaten.

Though more complicated configurations exist, the version that appeared on Nokia phones will be used. The playing field will be 11 units tall by 20 units wide consisting of 220 available spaces. The snake will initially begin in the lower left corner, facing right, with an initial length of 9 units. The maximum score possible for this configuration is equal to the amount of initially empty spaces. Therefore, the snake can eat $(220 - 9 = 211)$ pieces of food before filling up the entire playing

field. When this state is reached, the game terminates and a perfect game has been played.

Ehlis, who uses an identical configuration, notes that if the snake were to trace a simple pattern, a perfect score could be achieved by visiting every square in a giant loop. Using this pattern, all pieces of food would be eventually be eaten. This behavior is simple to program but very difficult for machine learning techniques to develop. However, the genetic programming solution discussed in Chapter I was successful in producing a pattern to obtain an optimal score.

The goal of this study will be to evaluate the performance of the temporal difference learning algorithm in generating a policy that maximizes the score.

B Overview of the Reinforcement Learning and the TD Lambda Algorithm

1 Reinforcement Learning

Reinforcement learning is an area of machine learning that attempts to produce policies that maximize a reward signal. Without prior knowledge, a learning agent must figure out, through multiple attempts, which actions lead to rewards and those that lead to failure. As Sutton and Barto state, "*These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.*"

Though, there is a feedback reward signal, it is important to understand that reinforcement learning is not *supervised learning*, a kind of technique that teaches agents to learn by presenting the learner with the correct answer for all inputs. In the case of games and other interactive problems, supervised learning is inadequate; it would be very difficult to have correct examples that reflect the desired player behavior that are representative of all potential situations. Reinforcement learning allows agents to learn though knowledge obtained through interaction with the environment. This allows the learning agents to adapt new strategies and policies

without an external source of expert information as with supervised learning.

One important aspect of reinforcement learning is balancing the trade-off between *exploration* and *exploitation*. An agent will obtain new information by exploring its environment. It will learn about new potentially greater rewards if it tries actions in situations that have not been explored. Therefore an agent with a greater drive for exploration will be more aggressive in seeking maximum rewards. An agent that prefers exploitation will be more conservative and exploit current knowledge to make decisions that lead to familiar rewards.

The dilemma of balancing between exploration and exploitation is that a more aggressive agent will try to seek higher reward at the cost of making mistakes. While the chance of an exploitive agent of making an incorrect decision is lower, the agent will not receive greater payout. A good strategy would be for an agent to try new actions to develop knowledge of potentially hidden rewards within its environment and then progressively begin to prefer the actions that lead to known rewards [9]. This trade-off can adversely affect the performance of the learning algorithm and should be carefully considered.

The reinforcement learning technique used in this paper will consist of these elements: the agent, the environment, the policy, the reward function, and a value function. The agent must be able to sample the game's environment in order to learn and make decisions. The kind of information that the agent receives from the environment is important, as not all information that could be given is relevant. Often the environment will supply its current state to the agent by selecting a set of features that are capable of representing its most important aspects [9].

A learning agent's entire goal is to produce an action policy that maximizes expected reward. The policy is the set of rules that dictate the actions taken by the agent toward achieving this goal. The policy is adjusted as the agent learns through feedback from the reward signal.

The reward function effectively determines what actions are good and bad for

the agent. In essence, the reward function defines the agent's overall purpose for interacting with the environment; it defines the motivations for the agent and provides true feedback in the form of success or failure. Based on the reward function, the agent makes adjustments to its policy and value function.

The value function is the agent's perceived expected future reward given the current state. It is not the actual reward signal, but more of an approximation of the reward function. It can be used to tell how well the agent is performing at any time. The value function is something that is also learned through exploration and therefore, can estimate an expected reward at any time during the learning process—even when the true reward signal is not present. By maximizing the value function, or the agents expected future returns, the agent can attempt to maximize rewards over the long-term.

Expected Rewards

The agent's goal is to maximize rewards over the long term. In the simplest case, the entire return is the sum of the sequence of future rewards obtained:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{1}$$

Where r_t denotes the reward given at time t. Sutton and Barto proposes that this would be useful for agents learning in finite time spans, called *episodes*. In cases where learning is continuous, the number of rewards may potentially be infinite. The process of *discounting* places a discounted value on future rewards with the γ parameter, called the *discount rate*. This has the effect of adjusting the worth of the reward at k time steps in the future to have γ^{k-1} times worth had if the future reward was not discounted.

The discounted return is defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

The γ parameter effectively sets the nearsightedness or farsightedness of the agent. An agent considering returns with $\gamma = 0$ would only seek to maximize immediate rewards; it would choose the action that would maximize the reward in the very next time step. It is generally accepted that seeking immediate rewards will reduce the possibility of obtaining better long term returns [9]. Adjusting the γ closer to 1 would motivate the agent to consider rewards more distant in the future. The choice of γ will be critically considered for the evaluation of the technique considered in this study.

C Temporal Difference Learning

Temporal Difference learning is a method of prediction. TD methods are useful for reinforcement learning problems as they can provide a way to define the value function. TD methods can be used to accurately predict the expected future reward. By using the difference of the predicted reward between successive time steps (hence the name *temporal difference* learning) each prediction is updated such that it will be made closer to the next successive prediction. When the reward signal is introduced, the value function will learn to estimate the expected sum of future rewards [8].

1 The TD Lambda Algorithm

To define the TD(λ) algorithm we will briefly define the following reinforcement learning symbols:

s_t**:** The current state at time t.

$V(s_t)$**:** The value function based on s_t that predicts expected reward.

π**:** The policy.

α**:** The learning rate.

γ**:** The discount factor for considering future rewards.

r_t**:** Observed reward at time t.

The simplest form of the TD(λ) algorithm is TD(0). The predication estimate at the current state is updated by the observed reward and the estimate at the next time step, by the update rule:

$$V(s) \leftarrow V(s) + \alpha\delta \tag{3}$$

$$\delta_t = [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{4}$$

Where δ_t is called the *TD error* (The TD error will be referenced extensively in this document.) The value function is adjusted such that the TD error is reduced; the current value function $V(s_t)$ is made to approach the discounted expected future reward, $\gamma V(s_{t+1})$ combined with any reward immediately obtained, r_{t+1}. The α parameter allows the value function to be updated as fraction of the TD error. Adjustments made in small increments are desirable. Please note that in equation 3, only the movement from the old state to the new state is considered responsible for the reward even though a succession of previous states could be responsible. This is fixed through TD(λ).

Eligibility Traces

The generalization of TD(0) to TD(λ) is through the incorporation of *eligibility traces*. The λ parameter adjusts the eligibility to which the sequence of past visited states can be attributed to the current reward. The eligibility trace is the coefficient to which a previous state is considered and is defined as:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & : s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & : s = s_t \end{cases} \tag{5}$$

This kind of trace is called an *accumulating trace* as the credit assigned to the state is accumulated each time the state is visited and decays each step the state is not visited; hence the reason λ parameter is called the *trace-decay parameter*. The effect of using λ is that more recent state-actions will be considered more influential in attaining the current reward than that of those that occurred further in the past; adjusting λ will change the degree an agent will consider how far in the past states are accountable for the reward signal generated [9]. The effect of changing λ credits the TD error to game states proportionally to their eligibility trace during learning. This is the backward facing view of the TD(λ) algorithm and is easy conceptually and in implementation [9].

When $\lambda = 0$, the agent will only credit the last action to the observed reward. Since the strategy of finding rewards usually requires visiting more than two states, it is important to credit reward to the past process of visited states. A non-zero value of $\lambda < 1$, will achieve this goal. A $\lambda = 1$ will consider all previous states equally in the credit assignment. For some problems, there is a time where states are so far in the past that they should not be considered. The λ parameter must be experimentally chosen for each learning problem.

2 The Definition of the TD Lambda Algorithm

As stated, the TD(λ) algorithm uses eligibility traces to credit past states to rewards. The update rule from equation 3 is adapted to include the trace-decay parameter, λ:

$$V(s_{t+1}) \leftarrow V(s_t) + \alpha\delta_t e(s_t) \tag{6}$$

The pseudocode for the TD(λ) algorithm is presented in Algorithm 1 [9].

Algorithm 1 TD Lambda Pseudocode

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
repeat {for each episode}
  $a \leftarrow$ action given by π for s
  Take action a, observe reward, r, and next state, s'
  $\delta \leftarrow r + \gamma V(s') - V(s)$
  $e(s) \leftarrow e(s) + 1$
  for all s **do**
    $V(s) \leftarrow V(s) + \alpha\delta e(s)$
    $e(s) \leftarrow \gamma\lambda e(s)$
  end for
  $s \leftarrow s'$
until s is terminal

D Function Approximation

Up to this point, there has been no mention as to how the value function, $V(s)$ is represented. The only information given is that it produces a scalar valued prediction from the current state, s. Function approximation is a way of defining $V(s)$ by assuming a model to generalize its outputs. Therefore, based on prior data, a well chosen model would learn to give reasonable predictions even when presented with states never before seen.

$V(s)$ can be generalized as a model or function that is defined by a set of modifiable parameters $\vec{\theta}$. As learning progresses, the output $V(s)$ can be modified by changing $\vec{\theta}$. An artificial neural network (See Section 3) for example, could define the model for mapping $s \rightarrow V(s)$ where $\vec{\theta}$ are the modifiable weights that determine the network's behavior.

1 Gradient-Descent

The model used for approximating $V(s)$ can be updated during learning by adjusting $\vec{\theta}$ such that it reduces the error between the predicted and actual returns. This is achieved by using the method of gradient-descent. After each observation, the model parameters are adjusted by a small amount in the direction that would

most reduce the error.

The simplest adjustment to $\vec{\theta}$ at the next time step would be:

$$\vec{\theta_{t+1}} = \vec{\theta_t} + \alpha[V^{\pi}(s_t) - V_t(s_t)] \nabla_{\vec{\theta_t}} V_t(s_t) \tag{7}$$

Where $V^{\pi}(s_t)$ is the actual expected return; $V_t(s_t)$ is the function approximation of $V^{\pi}(s_t)$ and the error is the difference between them. The $\nabla_{\vec{\theta_t}} V_t(s_t)$expression determines the direction of $\vec{\theta_t}$ that will minimize the error.

α defines the *step-size* parameter or *learn rate.* The learn rate should be chosen such that adjustments are made in small increments. If too large, the corrections will make $V_t(s_t)$ oscillate around the solution and if chosen too small, it will make the algorithm take a long time to converge to a solution.

In the specific case of TD(λ), equation 7 can be adapted to correct the TD error. The parameter adjustment will now consider eligibility traces and discounted returns:

$$\vec{\theta_{t+1}} = \vec{\theta_t} + \alpha \delta_t \vec{e_t} \tag{8}$$

Where δ_t is as defined in equation 4 and $\vec{e_t}$ is a column vector of eligibility traces that correspond to each model parameter in $\vec{\theta}$ and is updated by:

$$\vec{e_t} = \gamma \lambda \vec{e_{t-1}} + \nabla_{\vec{\theta_t}} V_t(s_t) \tag{9}$$

The gradient decent method for function approximation can easily be implemented. The pseudocode is presented in Algorithm 2.

This algorithm will be a key part of this study's implementation and evaluation.

2 Neural Network Review

Artificial neural networks are comprised of inter-connected computational units called neurons. These neurons individually transform a vectored input into a

Algorithm 2 TD Lambda Implemeted with Incremental Gradient-Descent

Initialize $\vec{theta}$ arbitrarily and $\vec{e} = 0$
repeat {for each episode}
  $s \leftarrow$ initial state of episode
  repeat {for each step of episode}
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
    $\vec{\theta} \leftarrow \vec{\theta}\alpha\delta\vec{e}$
    $s \leftarrow s'$
  until s is terminal
until end of episodes

scalar quantity. Each neuron has an input scaling coefficient, called *a weight* associated with each element of input. The linear combination of the input vector, $\vec{x}$, and the weight vector, $\vec{w}$, yields the neuron *activation*, z:

$$z = \sum_{i=1}^{n} w_i x_i \tag{10}$$

or in vector form,

$$z = \vec{w}^T \vec{x} \tag{11}$$

The activation of the neuron is then fed through the *activation function*, $\varphi(z)$ to get the final scalar output of the neuron o.

$$o = \varphi(z) \tag{12}$$

A graphical representation of a neuron is depicted in Figure 2.

There are several kinds of activation functions, though only two will be used in this study: *sigmoid* and *linear*.

The linear activation function simply forwards the weighted sum of the

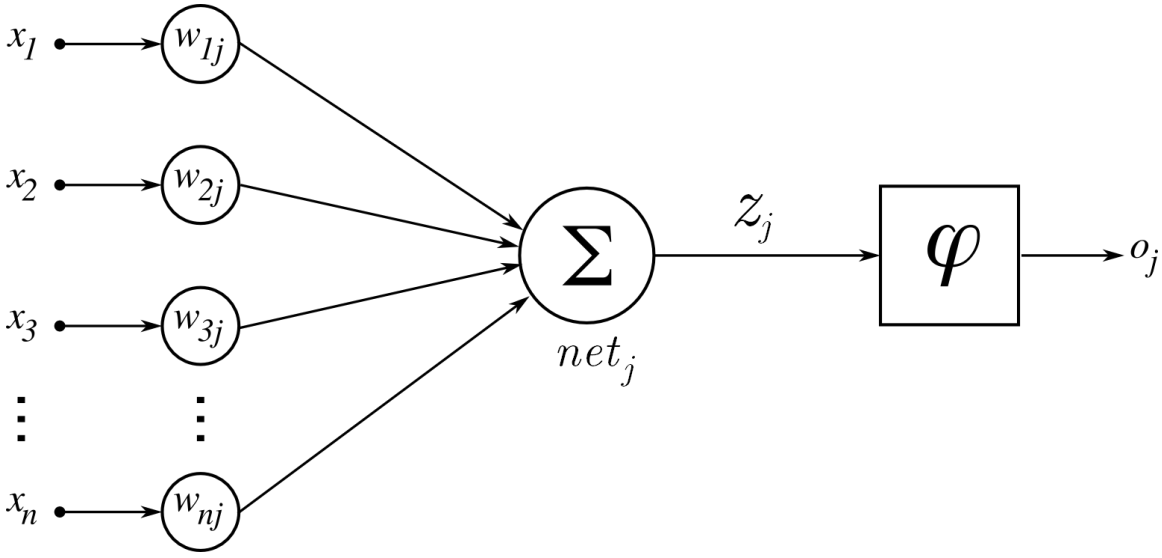


Figure 2. Artificial Neuron Model

inputs as the final output.

$$o = \varphi(z) = z \tag{13}$$

If used in the construction of an entire network, the linear combination of a set of linear combinations is still linear. Therefore, to introduce non-linearity into the results, a non-linear activation function must be used in the network [7].

The sigmoid activation function is a non-linear function that is bounded between either 0 to 1 or -1 to 1, depending on which specific sigmoid function is selected. Functions that are bounded in the range -1 to 1 will train more quickly as the network weights are better conditioned [7]. For this study, the hyperbolic tangent function will be used:

$$\varphi(z) = \tanh(z) \tag{14}$$

An example of the sigmoid activation function can be seen in figure 3.

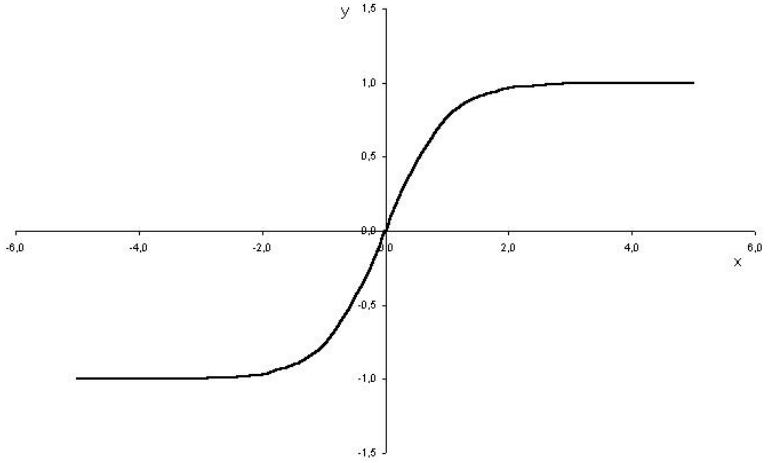


Figure 3. Plot of the hyperbolic tangent function: y=tanh(x)

The Multilayer Perceptron

Individual neurons can be combined in sets called layers. Layers of neurons can then be combined to create what is called the *multilayer perceptron* [5]. In a multilayer perceptron, data feeds forward from layer to layer; each output of a neuron in a layer is connected as an input to each neuron of the next layer. Neurons in the same layer cannot be connected together. This creates a fully interconnected web between layers (See figure 4 [1]).

The last layer of the perceptron is called the *output layer*. Any layers before the output layer are called *hidden layers*. A neuron inside a hidden layer is a *hidden neuron* [5]. Typical construction of a multilayer network places non-linear activation functions on hidden nodes and linear activation functions on the outputs. This is such that the output of the network can be the linear combination of non-linear features, and therefore, can produce any real valued number. Perceptrons in this configuration can learn to approximate any continuous function, given that there are enough nonlinear hidden units [6]. This configuration will be tested in this study.

For clarity, the following definitions are presented. A multilayer network will be described containing a number of layers L. The layers are numbered from the

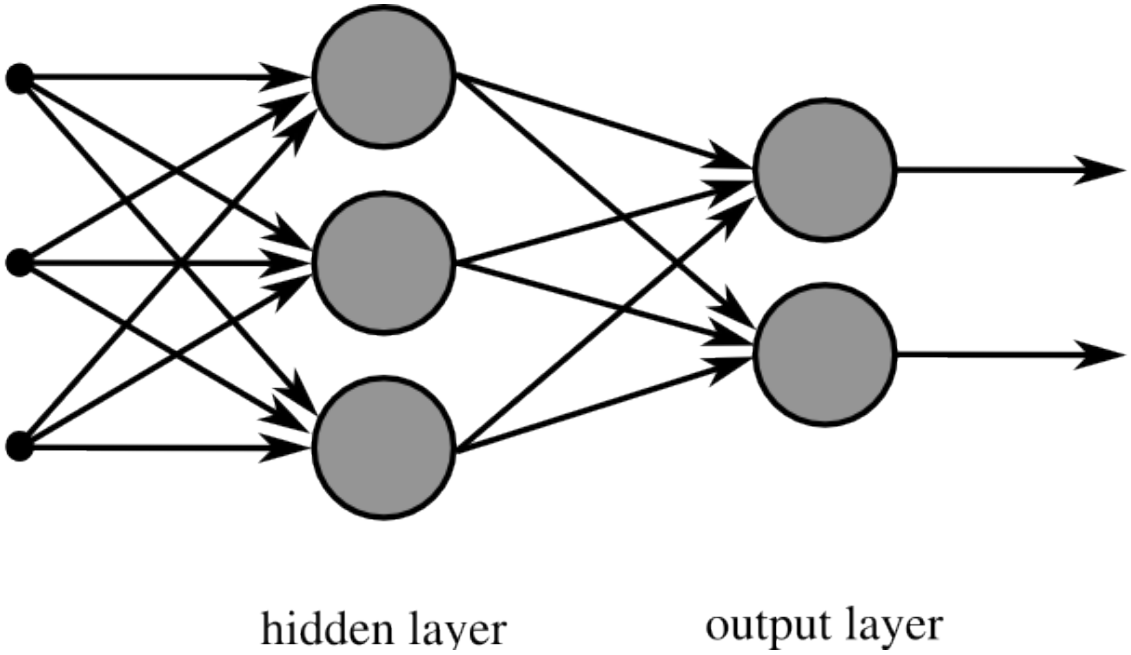


Figure 4. Multilayer Neural Network

input layer ($l = 1$) ascending to the output layer ($l = L$). Input data to a layer l is represented by $\vec{x}_l$ and the output of a layer is $\vec{o}_l$. Note that the output of one layer is the input to the next therefore, $\vec{x}_l = \vec{o}_{l-1}$.

Also, individual connections between layers can be identified by using a subscript. To denote the weight along the connection from neuron i to neuron j, w_{ji} will be used.

It is convenient to represent all of the weights that connect to neurons in a layer as a matrix W_l. Each column corresponds to the set of weights of a node in the layer. For a layer consisting of N units, the weight matrix can be formed by

$$W_l = \begin{bmatrix} \vec{w}_1^T \\ \vec{w}_2^T \\ \vec{w}_3^T \\ \cdots \\ \vec{w}_N^T \end{bmatrix} \tag{15}$$

To evaluate the network, data is presented to the network as the input to the first layer, $\vec{x}_1$ then output of each layer is computed in a feed forward pass from layer 1 to L,

For $l = 0$ to L...

$$\vec{o}_l = \varphi(W_l \vec{x}_l) \tag{16}$$

This equation assumes all nodes in the layer have the same activation function.

3 Value Function Approximation with Neural Networks

As stated, Multilayer Networks can approximate functions. In the case of reinforcement learning, neural networks can be a model for defining the value function, $V(s)$. Adjusting a network's model parameters, $\vec{\theta}$ is equivalent to adjusting the network weights. In fact, $\vec{\theta}$ is comprised of the network weights.

Gradient-Descent with the TD Lambda Algorithm

Recall that to derive an accurate value function, the model parameters must be adjusted such that they reduce the error of observations. In the case of neural networks, the weights must be adjusted appropriately. This occurs through the gradient calculation of $V(s)$ with respect to $\vec{\theta}$.

The calculation of the gradient for multilayer networks occurs in the well known process of *backpropagation*. This process finds the contribution of the networks weights to the error. The backpropagation algorithm can be adapted to operate on the TD error for training.

The Backpropagation Algorithm

Normally, the backpropagation algorithm is used in traditional supervised learning techniques where correct input/output data pairs are repeatedly given to the network. The computed error between the network's output and the desired

output is minimized by adjusting the weights of the network in the direction of the negative gradient. For multiple layers, the difficulty is in computing the proportion to which each node has contributed to the error; the error must be carefully propagated backwards from the output to inputs, hence the name, *backpropagation*.

The first step is to evaluate the network and record the outputs for all neurons. Then the contribution of error δ_j can be calculated for each node j in the network starting with units in the output layer and progressing towards the inputs. The error contribution is calculated differently for output units and hidden units. For the output units, the δ_j is calculated by

$$\delta_j = (o_j - t_j)\frac{\partial \varphi_j(z_j)}{\partial z_j} \tag{17}$$

and for the hidden units,

$$\delta_j = \frac{\partial \varphi_j(z)}{\partial z} \sum_{k \in l+1} \delta_k w_{kj} \tag{18}$$

Where $\frac{\partial \varphi_j(z)}{\partial z}$ is the partial derivative of the activation function with respect to its input.

In terms of layers, the set of δ_j's that correspond to the set of node in that layer can be represented by $\vec{\delta}_l$ and the above equations can be rewritten in terms l:

$$\vec{\delta}_l = (\vec{o}_l - \vec{t})\frac{\partial \varphi(z_l)}{\partial z_l} \ : l = L \tag{19}$$

$$\vec{\delta}_l = \frac{\partial \varphi(z_l)}{\partial z_l}(W_{l+1}^T \vec{\delta}_{l+1}) \ : l \neq L \tag{20}$$

Once the contribution of error for each node has been calculated, the weight of nodes for each layer can be adjusted proportionally by the equation,

$$w_{ji} = w_{ji} + \alpha \delta_j x_i \tag{21}$$

or in vector form,

$$W_l = W_l + \alpha \vec{\delta}_l \vec{x}_l^T \tag{22}$$

The pseudocode for the backpropagation algorithm is given in algorithm 3. For brevity, the derivation of the formulas was omitted. More information on the derivation of the backpropagation algorithm can be found in [6].

Algorithm 3 Backpropagation Algorithm

```
Initialize network weights, w, to small random values
repeat
  for all ⟨t⃗, X⃗⟩ training pairs do
    {Forward propagate the input X⃗}
    x⃗_1 ← X⃗
    o⃗_1 ← φ(w⃗_1^T x⃗_1)
    for l = 2 to L do
      x⃗_l ← o⃗_{l−1}
      o⃗_l ← φ(W_l x⃗_l)
    end for
    {Compute the error for the output layer}
    δ⃗_L ← t⃗ − o⃗_L
    {Compute the error for hidden layers}
    for l = L − 1 to 1 do
      δ⃗_l ← (∂φ(z_l)/∂z_l (W_{l+1}^T δ⃗_{l+1}))
    end for
    {Update weights in each layer}
    for l = 1 to L do
      W_l = W_l + α δ⃗_l x⃗_l^T
    end for
  end for
until error < threshold ∀⟨t⃗, X⃗⟩
```

4 Backpropagation of The TD Error

The backpropagation algorithm can easily be adapted for use in the TD(λ) algorithm. The necessary changes will be discussed.

Recall that TD(λ) procedure is on-line, that is to say that learning occurs with an agents interaction with the environment. This interaction is defined by the

current policy, π, and for this procedure, π does not change. The network will attempt to learn to predict the returns under π which is defined by $V^{\pi}(s)$. Remember that the agent's estimate of $V^{\pi}(s)$ is $V(s)$, therefore, the network is used to learn and evaluate $V(s)$; the network can be seen as the functional approximation of $V(s)$.

Since the network approximates $V(s)$, it must take state information as input to generate a prediction, we assume this information is generated by the environment and is comprised as a multidimensional vector of features. This vector will be used as the network input. The network will need only one output unit used to generate the prediction.

For use with TD(λ), the backpropagation procedure is initiated during the parameter update operation defined in equation 9. Equation 9 tells us how eligibility traces should be included to update the weights of the network. From this relation, we can update the backpropagation procedure for weight adjustment. The procedure outlined in equation 22 becomes:

$$W_l = W_l + \alpha \vec{\delta}_l \vec{e}_{lt}^{\,T} \tag{23}$$

where $\vec{e}_{lt}$ is the eligibility trace coefficients for each input in layer, l. It has been updated from equation 9 to become:

$$\vec{e}_{lt} = \gamma \lambda \vec{e}_{lt-1} + \vec{x}_l \tag{24}$$

The backpropagation algorithm used pairs of target outputs, $\vec{t}$ for specified inputs, $\vec{X}$ to calculate the error. In TD(λ), the error comes from **differences in temporal predictions**, by definition. Therefore, the error used for backpropagation in TD(λ), must come from the standard TD error, δ defined in equation 4. Formally, this modifies the definition for error assignment in the output layer of the network.

$$\delta_l = \delta \ : l = L \tag{25}$$

The error assignment for the hidden nodes is not changed; the TD error will be backpropagated correctly internally by the process defined in equations 18 and 20. The integration of iterative TD(λ) and backpropagation is presented in algorithm 4.

Algorithm 4 TD Lambda Implemented with Backpropagation

For all layers, initialize $\vec{w}_l$, to small random values and set $\vec{e}_l = 0$
repeat {for each episode}
  $s \leftarrow$ initial state of episode
  repeat {for each step of episode}
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\vec{x_1} \leftarrow s$
    Evaluate network for $V(s)$
    {Compute the error for the output layer}
    $\delta_t \leftarrow r + \gamma V(s') - V(s)$
    {Compute the error for hidden layers}
    for $l = L - 1$ to 1 **do**
      $\vec{\delta_l} \leftarrow (\frac{\partial \varphi(z_l)}{\partial z_l}(W_{l+1}^T \vec{\delta}_{l+1}))$
    end for
    {Update eligibility traces}
    for $l = 1$ to L **do**
      $\vec{e}_l \leftarrow \gamma\lambda\vec{e}_l + \vec{x}_l$
    end for
    {Update weights in each layer}
    for $l = 1$ to L **do**
      $W_l = W_l + \alpha\vec{\delta}_l\vec{e}_l^T$
    end for
    $s \leftarrow s'$
  until s is terminal
until end of episodes

E Bootstrapping and the Biased Estimate of the Value Function

Since the TD prediction is based upon a previous prediction (as in Equation 6) it is said to employ *bootstrapping*. The previous prediction could be wrong, but it is still used in the update process. Sutton and Barto show that while this may be the case; bootstrapping techniques do learn to make accurate predictions. Though there is one caveat: when bootstrapping is employed, the TD algorithm can only make a *biased estimate* of the value function, $V(s)$. A biased estimate is an estimate that is consistently an overestimate or underestimate. This implies that the TD prediction does not actually predict future returns, but it predicts future returns with a scaling factor.

Even though TD prediction is shown to be biased when bootstrapping is used, it still is very functional. Regardless of the bias, TD prediction can still predict very accurately which states are better than others. This is because learning comes from the relative difference in predictions and not from an absolute source.

Bootstrapping can be removed by removing the next state prediction whenever there is a reward signal generated. The TD update equation would become:

$$V(s_{t+1}) \leftarrow \begin{cases} R_t + \alpha\delta_t e(s_t) & R_t \neq 0 \\ V(s_t) + \alpha\delta_t e(s_t) & \text{otherwise} \end{cases}$$

F Random Walk: An Example

A random walk is a Markov chain with a finite number of sequential states that are occupied by a figurative entity called, the *walker*. At each state, the walker can only decide to walk to the left or to the right, in which case the walker transitions to the corresponding adjacent state. The decision to walk left or right, in this case, is made with equal probability. See Figure 5 for an illustration of the

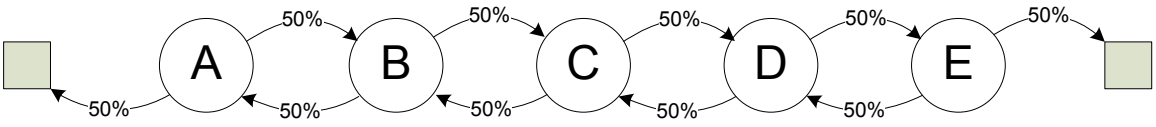


Figure 5. A diagram depicting the five state random walk used in the example. The walk begins at the center position and terminates when the walker attempts to transition out of the end states (depicted by the boxes.)

system. The walk begins by placing the walker into the center state and ends when the walker attempts to walk out of the states at the end of the chain. At the start, the probability of reaching the leftmost and rightmost states are equal but change as the walker drifts further from the center.

To demonstrate the prediction capabilities of the TD(λ) algorithm, a reward policy can be associated with the states of the walk such that the probability of transitioning out of the rightmost state can be predicted. If the walker chooses to transition out of the rightmost state, a reward of 1.0 is given, and for an attempt to transition out of the state that is furthest to the left, a reward of zero is given. This will allow the TD(λ) algorithm *using an unbiased estimate* to predict the expected reward for each state in the chain. Given this reward policy, the actual expected returns is equivalent to the probability of terminating the walk on the right. Therefore, it can be measured how well the TD(λ) algorithm is capable of predicting expected rewards by computing the Mean Squared Error (MSE) between the predictions and the actual probability of reward. For a five state random walk, the probability of reward is $\frac{1}{6}$, $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, and $\frac{5}{6}$ for each respective state in the system [9]. By varying λ and α, it can be seen if and how well the algorithm, using neural networks, can accurately predict future returns.

A two layer neural network was constructed with five hidden units to estimate the value function for a five state random walk without bootstrapping. The network was updated by the process outlined in Algorithm 4. Figure 6 shows the mean squared error of prediction after 100 episodes. The error is averaged over

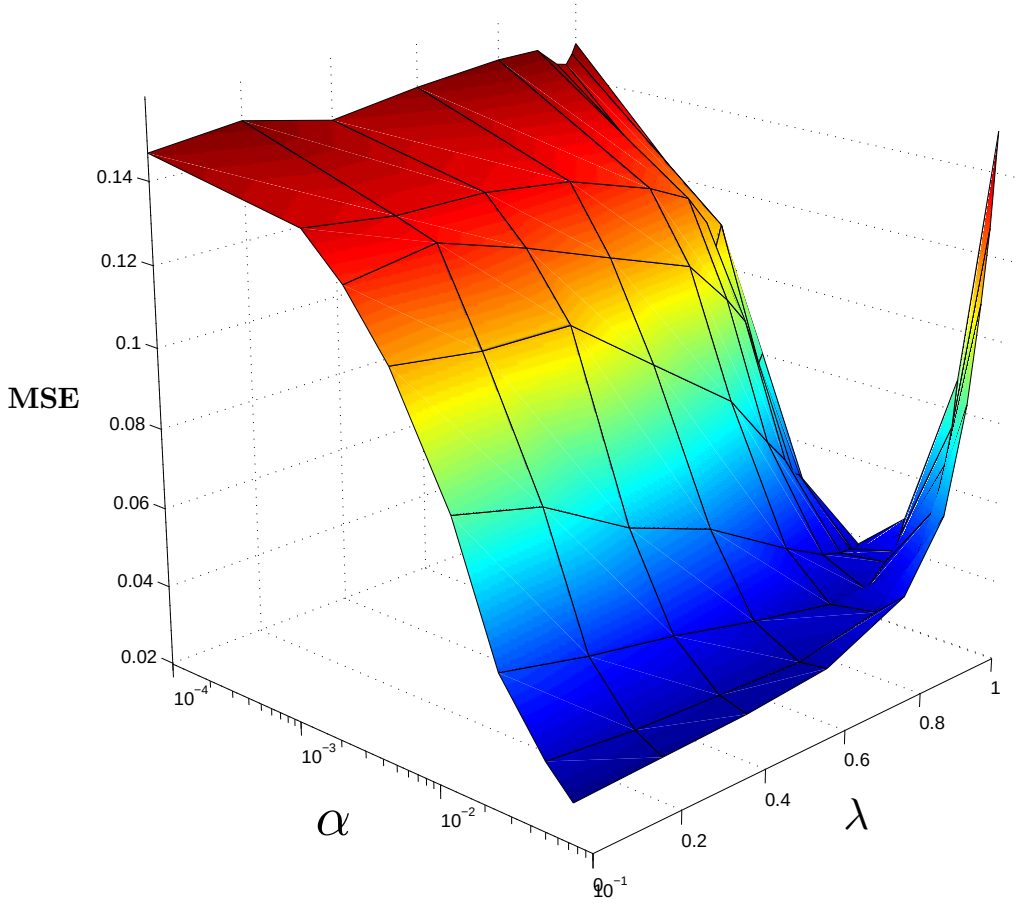


Figure 6. A surface plot of a neural network estimating the value function of a five state random walk for varying degrees of λ and α. Results were averaged over 1000 trials.

1000 trials. The graph shows for a given learn rate α, there is a choice for the trace decay parameter λ that minimizes the error. However, the variance is quite large for larger values of λ. See Figure 7.

The results do show that the backpropagation algorithm adjusted for use with the TD(λ), does in fact learn to predict the actual probabilities of reward. This verifies that using neural networks to approximate the value function in reinforcement learning problems is appropriate.

G Learning a Policy with the Actor-Critic Architecture

Up to this point, the definition of the agent's policy has been vague. It has been assumed that an agent interacts with its environment by choosing an action dictated by policy, π. The previous method introduced allows the agent to predict expected future reward without mention as of how an agent learns to make decisions.

In order for the agent to adapt, it must make changes to its policy. The actor-critic architecture is the tool that allows the agent to accomplish this task. The actor is the entity that will learn to make decisions, while the critic will learn to criticize them.

1 The Critic

From the information given, the critic is already well understood. The critic is the predictor of future returns (i.e. it is the incarnation of the $V(s)$ function.) As the agent makes decisions that move it from state to state, the critic learns to predict expected rewards. If the agent makes a decision that reduces the expected rewards, the agent should be less likely to try that action again given a similar situation. However, if the agent makes a decision that increases the predicted rewards, the action should be reinforced. It is the job of the critic to criticize the agent's actions by using the TD prediction method introduced.

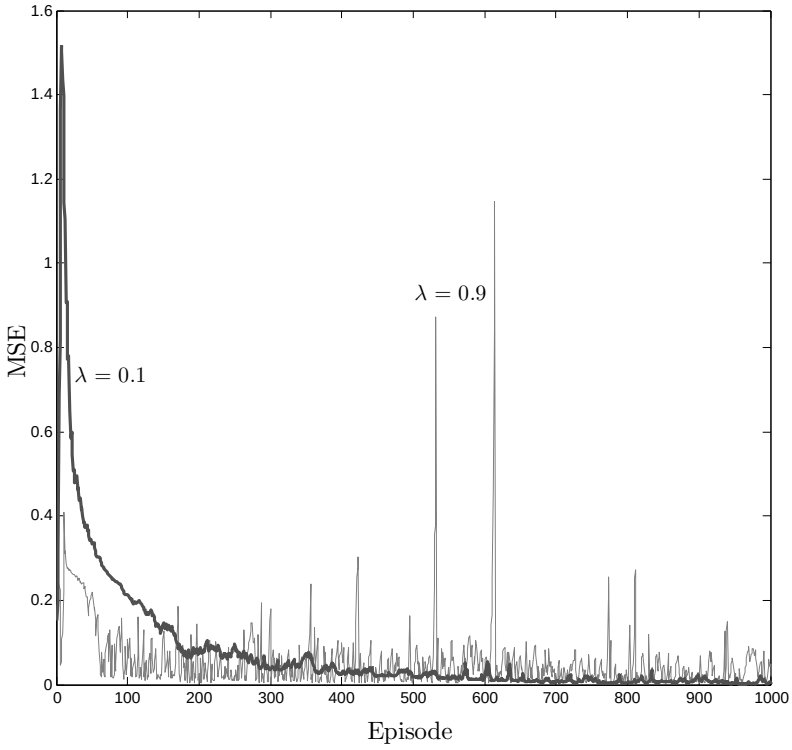


Figure 7. The Mean Squared Error of a two layer neural network estimating the value function of a five state random walk. The higher the trace decay parameter, the more noise is present.

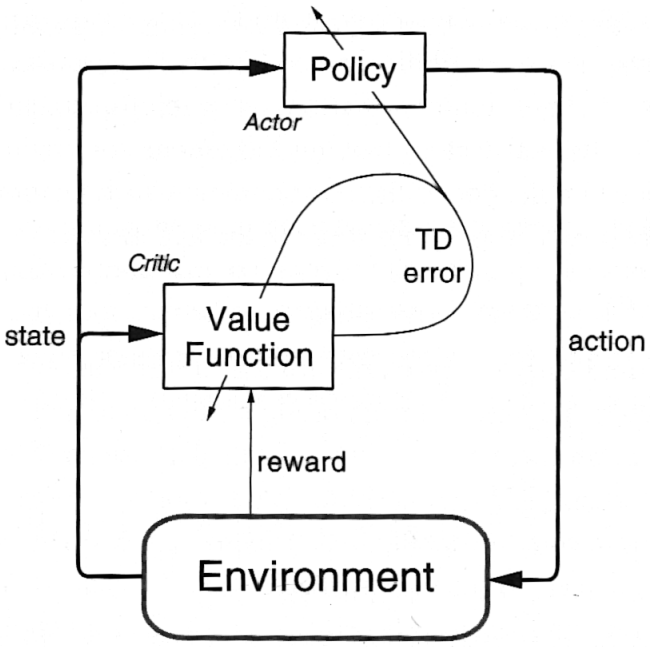


Figure 8. Actor-Critic Architecture

2 The Actor

The actor is what enacts the policy for the agent. The actor reads the current state of the environment and makes the decision to make an action. The actor, through criticism from the critic, will learn to make actions that lead to maximal rewards.

The actor learns its policy similar to the way the critic learns to make correct predictions. The critic learns to adjust its prediction through the TD error, or the difference in successive predictions. The critic will also use the TD error to adjust the actor's policy. When a state is entered, a prediction is made and reward or punishment may be presented. The action that was taken to achieve the current state should be blamed for the difference in successive predctions (i.e. the action should be blamed for the TD error). If the reward or the prediction estimate improved, the TD error will be positive and it can be used to reinforce the actions. The described architecture along with the path of the TD error is shown in figure 8.

Adjusting the Policy With Eligibility Traces

The adjustment to the actor policy should follow the procedure formally stated by Sutton and Barto.

$$p(a, s) \leftarrow p(a, s) + \beta\delta_t \tag{26}$$

Where $p(a, s)$ is the probability of selecting action a while in state s. It is critical that when adjusting the policy of the actor, that the appropriate state-action probabilities are adjusted. The concept of eligibility traces can be expanded to affect the action-state pairs of the actor. This allows the assignment of the TD error to not only the past states, but to the sequence of actions that led to the current reward.

When incorporating state-action eligibility traces, the update rule from equation 26 becomes:

$$p(a, s) \leftarrow p(a, s) + \beta\delta_t e(s, a) \tag{27}$$

where $e(s, a)$ is incremented by one every time s and a appear together.

In this thesis, a proposal to update the policy by applying eligibility traces to the actor is given in Chapter H.

Choosing the Action: Balancing Exploration and Exploitation

The actor chooses the next action by specifying a preference for each action. The preference is a scalar value representing how strongly the actor considers the action to be good. $Q(a)$ is called the *action-value function* and will represent the real valued preference for action a.

An entirely exploitative policy will select the action of the highest preference. This is because the actor will use known strategies and will resist trying new moves that may lead to better rewards. Therefore, randomness must be introduced in

action selection to allow the actor to explore. This is facilitated by SoftMax probability selection [9].

In order to choose actions that may lead to undiscovered rewards, an action that might not have been preferred should be taken. If an action leads to greater returns, its preference for that action will be increased and thus, the action will be selected with a greater probability given a similar opportunity. SoftMax selection places a probability on each action based on the actor's preferences. The actor then randomly selects the next action by the calculated probabilities. The selection of less probable actions may lead to undiscovered rewards [9].

A parameter τ, called the *temperature*, adjusts the probability calculation and effectively controls the extent to which the agent is exploitative or explorative. If $\tau = 0$, the agent is purely exploitative and will guarantee a selection of the action with the highest preference. As τ gets large, the probabilities are all equally assigned, making the effective choice of action entirely random [9].

The SoftMax selection equation will assign action, a, from n actions with probability, p_a:

$$p_a = \frac{e^{\frac{Q(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q(a)}{\tau}}} \tag{28}$$

The choice of τ in this study will be critically examined.

H Proposal for Actor Eligibility Traces

1 Neural Network Actor

A practical implementation of the actor's policy can be made through a multilayer neural network. The network would have an output unit for every action. Based on the input state, the network will output it's preference for each action, effectively evaluating the action-value function Q at the outputs. The action preferences can be combined into vector $\vec{q}$ and the action taken will be determined

by the SoftMax selection.

Adjusting the Policy with Eligibility Traces

The adjustment of the actor's policy will be based on the TD error produced by the critic. The policy should be adjusted to affect not only the states, but the actions that caused the error. Therefore, credit will be assigned to both states and actions during the policy update. For a multilayer network, the contribution of an action to the TD error must be computed. The literature regarding the process of incorporating actor eligibility traces is vague in terms of neural networks. Therefore, I propose a method for integrating action based eligibility traces in a multilayer neural network.

The Methodology

The eligibility for weight adjustment in the network will contain two parts: the eligibility from the action selected, and the eligibility from the state visited. Both eligibilities will be tracked separately. For state eligibility, the actor will keep eligibility traces in the same manner as the multilayer critic. The action traces will be kept in a vector, $\vec{e}_a$ whose elements correspond to the units in the output layer.

The selected action from the SoftMax process is presented to the actor network in the form of a unit vector $\vec{a}_s$, called the *action selection* vector, where each element is zero except for the element corresponding to the selected action. For example, if the first action in a policy of three were selected, $\vec{a}_s$ would look like:

$$\vec{a}_s = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

The eligibility vector can now be defined and updated by the relation

$$\vec{e}_a \leftarrow \lambda\gamma\vec{e}_a + \vec{a}_s \tag{29}$$

$\vec{e}_a$ is used to assign the TD error, δ_t, proportionally to the neurons and weights that contribute to the actions responsible for it. For example, the j^{th} node in the output layer is directly responsible for selecting action j. If the eligibility for action, j, is non zero, that action and the output of unit j, contributed to the error *in the amount proportional* to the j^{th} eligibility coefficient. Therefore, the error will be backpropagated only along the paths that contribute to the j^{th} action being selected.

The backpropagation algorithm can be adapted to adjust the weights such that the TD error is assigned appropriately. Instead, of applying the error equally across all output units (as defined in the backpropagation algorithm) the TD error will be assigned to output units proportionally by $\vec{e}_a$. The error assignment for output units (from equation 25) becomes

$$\delta_l = \delta_t\vec{e}_a \ : l = L \tag{30}$$

The algorithm for the multilayer actor update is presented in algorithm 5.

Since the eligibilities for state and actions are tracked separately, they multiply together during the backpropagation process to form the final trace. Even though state eligibilities are updated the same as in equation 24, weight adjustments will be made only if both the state **and** action are eligible together. For example, even though a specific network weight may be eligible for adjustment from previous states, it will not be adjusted for actions whose eligibility is zero; there will be zero TD error backpropagated from the action neuron to the weight. This experimental procedure will be implemented in this study.

Algorithm 5 Update Pass for Multilayer Actor

Evaluate action prefrences $\vec{q}$ from s
Compute $\vec{a}_s$ from $\vec{q}$ using SoftMax selection.
Take action, observe reward, and next state, s'
{Compute action eligibilities}
$\vec{e}_a \leftarrow \lambda\gamma\vec{e}_a + \vec{a}_s$
Receive δ_t from critic
$\delta_o \leftarrow \delta_t\vec{e}_a$
{Compute the error for hidden layers}
for $l = L - 1$ to 1 **do**
  $\vec{\delta}_l \leftarrow (\frac{\partial\varphi(z_l)}{\partial z_l}(W_{l+1}^T\vec{\delta}_{l+1}))$
end for
{Update state eligibility traces}
for $l = 1$ to L **do**
  $\vec{e}_l \leftarrow \gamma\lambda\vec{e}_l + \vec{x}_l$
end for
{Update weights in each layer}
for $l = 1$ to L **do**
  $W_l = W_l + \alpha\vec{\delta}_l\vec{e}_l^T$
end for

CHAPTER III

DESIGN AND IMPLEMENTATION

This chapter will discuss the design considerations of using the proposed TD technique in an attempt to implement a computer agent capable of learning to play the game of Snake. In addition, an experiment will be designed to find the configuration of parameters and system factors that maximize the agent's learning performance. From the analysis of quantifiable performance metrics, the best Snake playing agent identified will be compared directly to the optimal performing solution as proposed by Ehlis.

The following sections outline the implementation, considerations and configurations for specific system components used for creating the agent and experiment.

A The Environment

The environment is a very important element of the reinforcement learning system. For this problem, the game serves as the environment. The game will be responsible for generating the reward signal. The reward signal, as discussed, defines the goal and purpose of the agent. A reasonable reward policy must be chosen such that the snake will be motivated to obtain the highest score.

1 Selecting a Reward Policy

There are two immediately identifiable skills that are useful for playing the game of Snake. These skills will be quantified by following the proposed

performance metrics; each will be considered when selecting a reward policy that encourages the agent to master them.

Staying alive: The first challenge for the agent is to avoid ending game by colliding with itself or the walls. Therefore, in the search for seeking the greatest returns, the agent should be rewarded for every time step the agent stayed alive, or equivalently, the agent should be punished with a negative reward if the agent terminates the game. Either way, the agent will seek the maximum reward.

Collecting Food: After staying alive, the secondary task of the player is to collect food. While not critical to the agent's survival, food collection increases the score of the game which after all, is the ultimate goal. A reward signal must be given to motivate the snake to achieve a perfect score. Seeking maximum returns, a reward given when the snake collects the food will achieve this goal. It is not intuitive, but it is possible that the agent may have trouble in identifying the reward associated with the food. This is due to the fact that the agent must accidentally consume the food to learn that the reward exists. For cases where agents develop *circling* behavior, the agent may never find the food piece as it prefers to follow a well defined circle. In this case, the agent has maximized known rewards by minimizing the number of negative rewards from dying. An agent stuck in a circle may never accidentally wander onto the food piece, especially if the agent is purely exploitative. Therefore, the agent must be encouraged to explore its environment. This can be done by the SoftMax action selection temperature, size of the board, and the reward policy. Circling behavior will be discouraged when the agent has gone a finite number of steps without receiving an award. This rule has shown to be very effective, as described in Chapter IV.

Based on the two challenges presented by the game, a reward policy can be made.

- A reward of -1.0 will be given when the agent terminates the game.
- A reward of $+1.0$ will be given when a food piece is eaten. This is such that the total rewards accumulated will be equal to the score obtained game. A perfect game will accumulate 211 units of reward.
- A reward of -1.0 will be given when the snake travels twice the number of spaces contained in the area of the board without eating a food piece. This will encourage exploration and discourage circling.

2 Considering Discounted Returns

In chapter 2, the concept of discounted rewards were introduced. Recall that the γ parameter controls the degree to which an agent seeks future rewards. A $\gamma = 0$, is myopic and prefers to select actions leading only to immediate rewards. Whereas a $\gamma = 1$ considers all future reward equally. Sutton and Barto suggest that the discount parameter is used for training sessions that may never end. Since the critic is attempting to predict the expected future reward in tasks that could receive a potentially infinite number of rewards, the critic could never actually learn to predict the accumulated sum of returns. The discount parameter can convert the infinite sum of rewards into a finite number by discounting future returns in the same way the inflation rate discounts the future value of the dollar.

In the case of snake, each training session is bounded in episodes and has a finite number of rewards. In this case, a critic is capable of predicting the actual accumulated reward. From the definition of the reward policy, the total number of rewards is bounded to 211. Therefore, having a $\gamma = 1$ is appropriate in this problem. For the study, the agent will not discount future returns and the algorithms will be implemented with a $\gamma = 1$.

3 Playing Field Size

The standard playing field for the game of Snake is 20 units by 11 units. Because this is a large area, the agent may take a long time to explore the field before it accidentally finds the food. Thus, to encourage the probability of the agent accidentally consuming the food and to reduce the time of training, the experiment will conduct trials with a playing field of reduced size. This assumes that the behaviors and skills learned in the smaller area will transfer to the larger field. For the experiment, training will be performed on a board that has a width and height of 8 units. When training is finished, an agent's performance can be assessed on the standard playing field.

4 State Representation

One of the considerations in constructing a Snake playing game agent is how the agent will perceive the environment. The environment must supply a set of features that can accurately represent the state of the game. These features are extracted at each time step and given to the actor and critic as a state vector $\vec{s}$ to produce the value estimate $V(s)$ and the next action, $\vec{a}_s$.

The Curse of Dimensionality

The curse of dimensionality refers to the exponential growth of a hyper volume as a function of dimension [7]. In neural networks, a large input vector may degrade performance. This is due to a couple of reasons. First, the network must sort out how the many variables interact and map to the output. This takes more time when more inputs are added. In fact, the extra time needed to train may grow non-linearly as the volume capable of representation grows exponentially.

Second, the more input elements, the greater the probability that they are irrelevant features. If so, the network must take extra time to learn that the space introduced by these features is unimportant. In addition, since the dimensionality is

high, the network will use much of its resources to represent the irrelevant space, robbing resources from features that need them.

Using this information, $\vec{s}$ for the game of Snake can be considered.

Visual State

Visual state representation attempts to present the information from the game to the agent in a way that is similar to how a human would perceive the game. The state vector $\vec{s}$ can be seen as a square 2-D image with each feature representing the contents of a location on the game board. At each time step, $\vec{s}$ would be populated by extracting the elements in the grid around the snake's head and placing a mapped ordinal value into the feature vector at the respective positions. For a location containing a wall or tail piece, the element would be mapped to -1. The food location would be +1 and empty positions, 0.

The image vector would have a fixed size and would be placed such that it is centered around the snake's head. For this experiment the sizes of the viewing window around the player's position will be (5 by 5) and (7 by 7). This will allow the snake to *see* 2 and 3 positions in every direction, respectively. While the agent cannot see the whole board using this technique, it can see the immediate area around the player's position.

This state representation has the advantage of allowing the agent to interpret the configuration of the snake's tail to some degree. However, it also has the disadvantage of potentially introducing irrelevant information. Perhaps not all locations around the snake's head are important to perfect play. The experiment performed by Ehlis proves that some of these visual features are not necessary for obtaining a perfect score. A state representation based on the features introduced in his experiment will also be tested.

It is hypothesized that the visual state representation will introduce interesting behavior, perhaps even human like. However, it is believed from the

available information that other state representations may lead to an optimal policy.

State of Important Features

Ehlis used a set of features for deriving a machine learning algorithm capable of playing a perfect game. Because this soultion performed so well, a state representation will be based on this set. For the experiment, Ehlis created features that observed the distance from the snake's position to danger(walls or tail) and to the food. Actually, only the presence of danger or food could be queried at specific directions and lengths around the head, but the agent could get accurate distance information by making multiple queries.

An adapted state vector for this study, will provide the agent with a state vector with five logical elements (taking only values of 1.0, or 0.0).

Danger Left: Indicates danger in the space to the left.

Danger Forward: Indicates danger in the space ahead.

Danger Right: Indicates danger in the space to the right.

Food Left: Indicates that food is somewhere in the row of spaces to the left only if danger is not between the food and the snake's head.

Food Ahead: Indicates that food is somewhere in the row of spaces in the forward direction if danger is not in the way.

Food Right: Indicates that food is somewhere in the row of spaces to the right only if danger is not in the way.

Note that the orientation is aligned with the controls that the snake will be using; it is relative to the player's direction of travel.

This feature set should be just as descriptive as the set initially used by Ehlis. In the article, Ehlis used two sets. The initial set was not as capable of

achieving good performance as the second set. Therefore, in order to replicate the same features, the second set features will also be used: they include all the features of the first set but also include direction and extra danger queries.

Danger Left Two: Indicates danger in the space two units to the left.

Danger Forward Two: Indicates danger in the space two units ahead.

Danger Right Two: Indicates danger in the space two units to the right.

Moving Up: Indicates that the snake is moving upward in the playing field.

Moving Down: Indicates that the snake is moving downward in the playing field.

Moving Left: Indicates that the snake is moving left in the playing field.

Moving Right: Indicates that the snake is moving right in the playing field.

The extra features may help facilitate beneficial behavior like zigzagging and other space filling techniques.

Combination of State Representations

A combination of the important state and the visual state will be tested. This will attempt to capture the same (if not more) information as proposed by Ehlis' second feature set. The visual state feature vector $\vec{s}$ will be expanded to include the food sensing abilities. This will make the agent capable of detecting the food, when it is not in its field of view.

B Building the Agent

The agent will be implemented using the actor-critic architecture as proposed in previous the chapters.

1 Implementing the Critic

Critic Interface

The critic must learn to predict the total expected future reward by observing $\vec{s}$. At each time step, the game will supply $\vec{s}$ and the critic will output its current prediction. The predictions of the critic between successive time steps will generate the TD error. The error signal will be used to adjust the critic and the actor.

Structure

A multilayer neural network with weights adjusted as described in the TD(λ) algorithm with backpropagation (Algorithm 4) will be used for approximating $V(s)$. The network will be constructed such that the output layer will contain one linear node responsible for producing the predicted reward. The best number of layers and neurons per layer will be experimentally determined. All of the hidden nodes will possess the hyperbolic tangent activation functions to capture any non-linear behavior required to control the snake. The number of inputs will be adjusted for the kind of state representation being tested.

2 Implementing the Actor

Actor-Environment Interface

For Snake, the actor will have to choose one of the three actions: move forward, turn left, or turn right. A multilayer neural network with three output nodes will be used to implement the policy of the actor. Each output will produce the actor's preference for each respective action. Based on the magnitude difference between each action and the SoftMax temperature, the actor will choose the action to take given the current environment state.

Structure

The actor will be trained using the novel adjustment procedure proposed in chapter II. Like the critic, the actor network will have its structure experimentally determined. The configuration of the number of nodes per layer and the number of layers will be tested to see which combination maximzes the agent's learning performance.

CHAPTER IV

EXPERIMENT AND RESULTS

An experiment will be conducted to find the best performing Snake playing computer agent. In the following sections, the construction of the evaluation experiment will be discussed.

A Tested Factors

As mentioned, there are several factors which are assumed to affect the performance of the learning technique. The following factors are suspected to have an effect on an agent's learning performance and will be tested.

α**:** The learning rate of the actor and critic.

λ**:** The decay parameter of the eligibility traces.

τ**:** The SoftMax temperature parameter used to select actions from the preferences provided by the actor. Effectively, it controls the balance between exploration-exploitation.

$\vec{s}$**:** Variations on the environment state representation.

$NeuralNetworkStructure$**:** The number of hidden nodes per layer and the number of layers in the actor and critic. These features can affect the quality of function approximation of these entities.

Quantitative measures for evaluating the performance of the learning agent must be created to assess any improvements made by changes in the input factors.

B Evaluating Performance with Quantitative Measures

Two measures can be constructed that evaluate an agent's ability to play the game of Snake. Each measure directly addresses the challenges of game play.

Average Length of Episode: As one of the challenges is to avoid ending the game, this metric characterizes how well the agent is able to navigate and avoid obstacles. It can be defined by the number of iterations successfully survived divided by the number of games played.

$$\frac{n_i}{N_g}$$

As an agent learns, it is expected that it will be better capable of surviving. Therefore, an increase in this measure is expected over time.

Average Score per Game: This metric directly assesses the agent's ability to complete the game's objective of collecting food pieces. The higher the average score per game, the better capable the agent is of surviving and collecting food. It can be defined as the number of food pieces eaten divided by the number of games played.

$$\frac{n_f}{N_g}$$

Both metrics attempt to quantify the distinct skills needed to play the game of Snake. An agent that plays the game well, will show that both metrics are high; the agent has learned to navigate and avoid obstacles and, has learned to do so while increasing the score. Thus, there are implications when an agent develops with unbalanced parameters. If an agent learns to maximize the length of episode without maximizing the rewards per episode, this implies that the agent does not care about the food and has preferred a policy that keeps it from dying. This can

happen for a few speculated reasons. The first: for some network configurations, it is possible that the agent may not have the capability to learn to do both tasks simultaneously. Learning to collect food may overwrite previously learned navigation skills. Because both skills cannot persist at the same time, the agent maximizes reward by collecting the fewest negative rewards.

In the second case, the agent may not identify that the food is good due to poor exploration. An agent that has learned to circle, never trying new things, may be satisfied because it has never correlated food reward. The agent must explore or accidentally hit the food to find out that it leads to greater returns. It is possible, through multiple failures, that learning updates overwrite previously learned knowledge where food seeking behavior could be unlearned.

The case where the agent receives an abnormally high proportion of rewards to the length of game is not possible as the minimum number of steps per food is defined by the minimum distance needed to navigate from food piece to food piece. Therefore, an agent with shorter game durations with higher rewards has learned to be more efficient at food gathering.

These metrics will be used to track the progress of a learning agent. Since the performance of the agent is expected to improve, it will be tracked and depicted in graphs called learning curves. This will allow the agent's performance growth to be assessed over time.

C Trials

Trials will be conducted to find the configuration of modifiable parameters that lead to peak learning. The suspected factors will be tested in terms of the performance metrics outlined in the previous section. An outline for factor testing is presented:

For each trial, the levels for the various factors will be set and training will begin by starting a new game. The agent will learn on-line, that is it will learn as

the game progresses. A new actor and critic will be created for the start of each trial. The trial will continue for a fixed number of games. Once the trial has ended, performance of the agent, as determined by the performance measures, will be recorded with the factor settings. The observation of significant performance improvement, over time, will indicate that learning has occurred. Therefore, the performance will be charted such that an analysis of the parameters on the effect of learning can be performed. These charts, again called learning curves, will be presented with the trial results in the upcoming sections.

Due to the large amount of factors, the experiment trials will be completed in sets. Each set will attempt to only vary factors that influence independent parts of the system. This will attempt to minimize the risk of interaction between factors in separate sets. Each set can be thought of as addressing a different aspect of the learning technique. The sets of trials are enumerated:

1. Learning parameters are adjusted. The learn rate and the trace decay parameters will be tested for the best configuration, given fixed levels for other factors.

2. Using the best configuration of the previous trial, the number of hidden nodes and number of layers in the actor and critic will be adjusted to seek any additional improvements.

3. Using the best configuration of the previous trials, the actor's SoftMax action selection temperature will be adjusted to change the balance between exploration and exploitation. The changes will be observed to seek any additional improvements in learning performance.

4. Using the best configuration of the previous trials, a variation of state representations will be used to seek any additional improvements in performance.

TABLE 1

Agent reward averaged over the first 1000 episodes.

Average Reward Per Episode					
λ	$\alpha = 0.0001$	$\alpha = 0.0005$	$\alpha = 0.001$	$\alpha = 0.01$	$\alpha = 0.1$
0.00	-0.76	4.74	8.24	4.20	-0.19
0.01	-0.75	4.01	9.22	3.95	-0.01
0.05	-0.01	5.73	6.86	2.61	-0.25
0.15	-0.47	4.53	6.13	2.62	-0.36
0.30	-0.24	3.12	3.33	1.61	-0.79

D Results

The results of the enumerated trial sets for the experiment outlined in Chapter III are presented with observations in the following sections.

1 Trial Set One

In this set of trials, the learn rate, α, and the trace decay parameter, λ, were varied to observe any effect on the agent's ability to learn to play the game. Other than these parameters, all agents were uniformly constructed. For each configuration, the actor and critic were both created with two hidden layers of 50 units. The actor was configured to select actions with a SoftMax temperature of 1.0. The important feature state representation was used for informing the agent of the environment state.

Agent performance data was collected for several configurations of of α and λ. In each trial, the parameters held at specific levels were tested with the number of rewards and length of each game tracked as the agents played 1000 games. Each trial was repeated 10 times. The average number of rewards per episode and the average episode length for each configuration is shown in Tables 1 and 2.

TABLE 2

The length of a game experienced by the agent averaged over the first 1000 episodes.

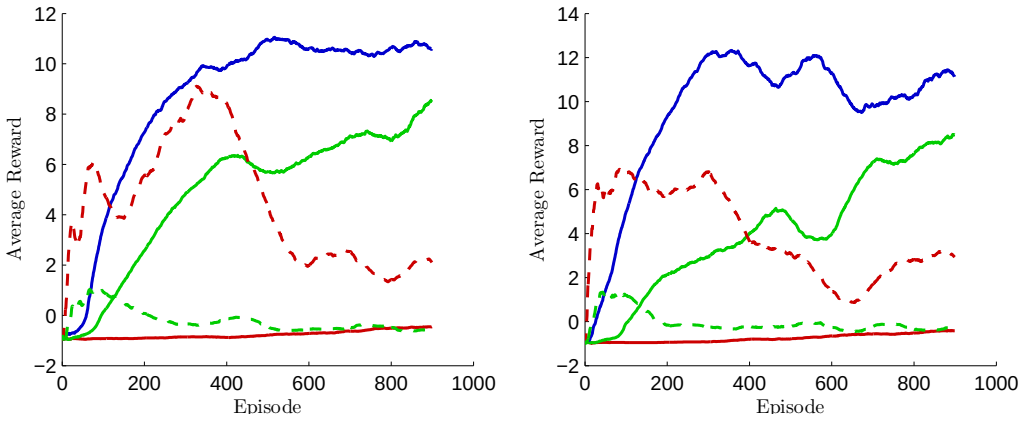
Average Length of Episode					
λ	$\alpha = 0.0001$	$\alpha = 0.0005$	$\alpha = 0.001$	$\alpha = 0.01$	$\alpha = 0.1$
0.00	14.08	123.54	144.25	87.39	42.35
0.01	14.90	96.18	143.46	92.04	37.36
0.05	26.97	119.37	126.10	96.62	33.24
0.15	22.69	98.86	102.05	100.1	43.83
0.30	26.84	91.97	102.69	86.56	13.90

In addition, the average learning curves for the tested configurations are presented in Figures 9, 10, 11, and 12. The curves show how the performance of the agent changes over time (as more games are completed.) For ease of interpretation, the data is presented in two sets of graphs. Both sets depict the same data but are constructed such that direct comparisons between different levels α or λ can be made. Figures 9 and 10 show how the learning curves differ by variations in λ with α held constant. Figures 11 and 12 allow direct comparisons of changes in α with λ held constant.

Choosing different values for α and λ had a significant effect on the agent's performance.

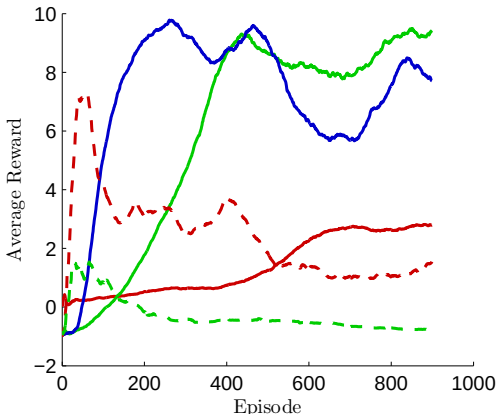
Correlation of Reward and the Length of Game

In all cases, the number of accumulated rewards per game and the game length were strongly correlated. Trends in the average reward are almost exactly reciprocated in the trends in average episode length for the same factor levels. This shows that both performance metrics capture the same quality of the agent's ability to play the game of Snake, and therefore, each individually can be used to fully

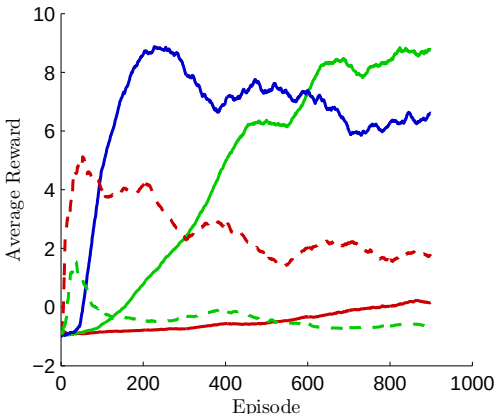


(a) $\lambda = 0.0$

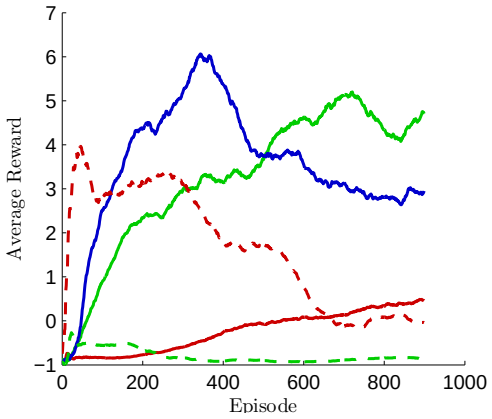
(b) $\lambda = 0.01$



(c) $\lambda = 0.05$



(d) $\lambda = 0.15$



(e) $\lambda = 0.30$

Figure 9. The effect of learning parameters on the average performance of the agent in accumulating rewards, over time, is depicted in each graph, grouped by λ.

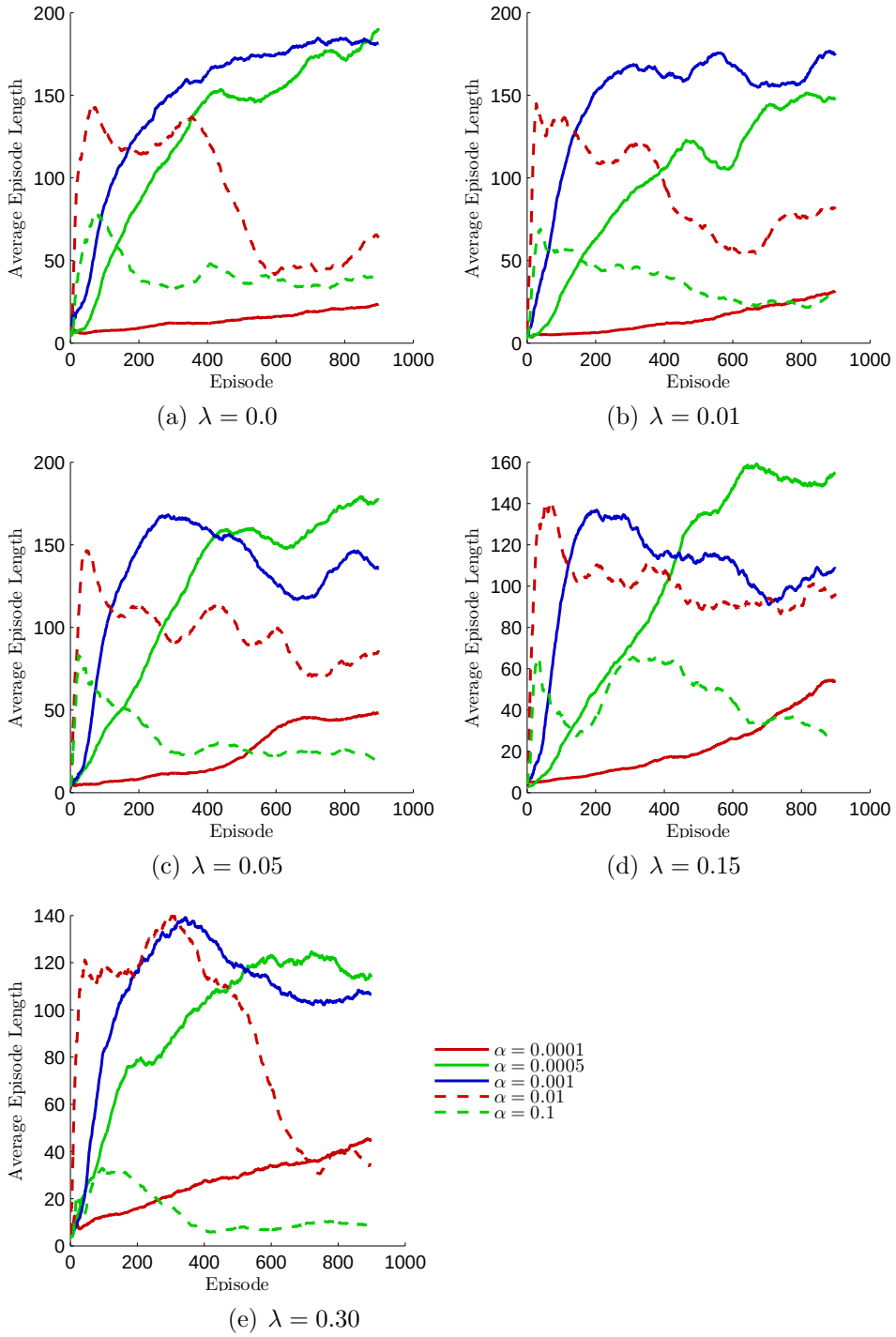


Figure 10. The effect of learning parameters on the agent's average length of game, over time, is depicted in each graph, grouped by λ.

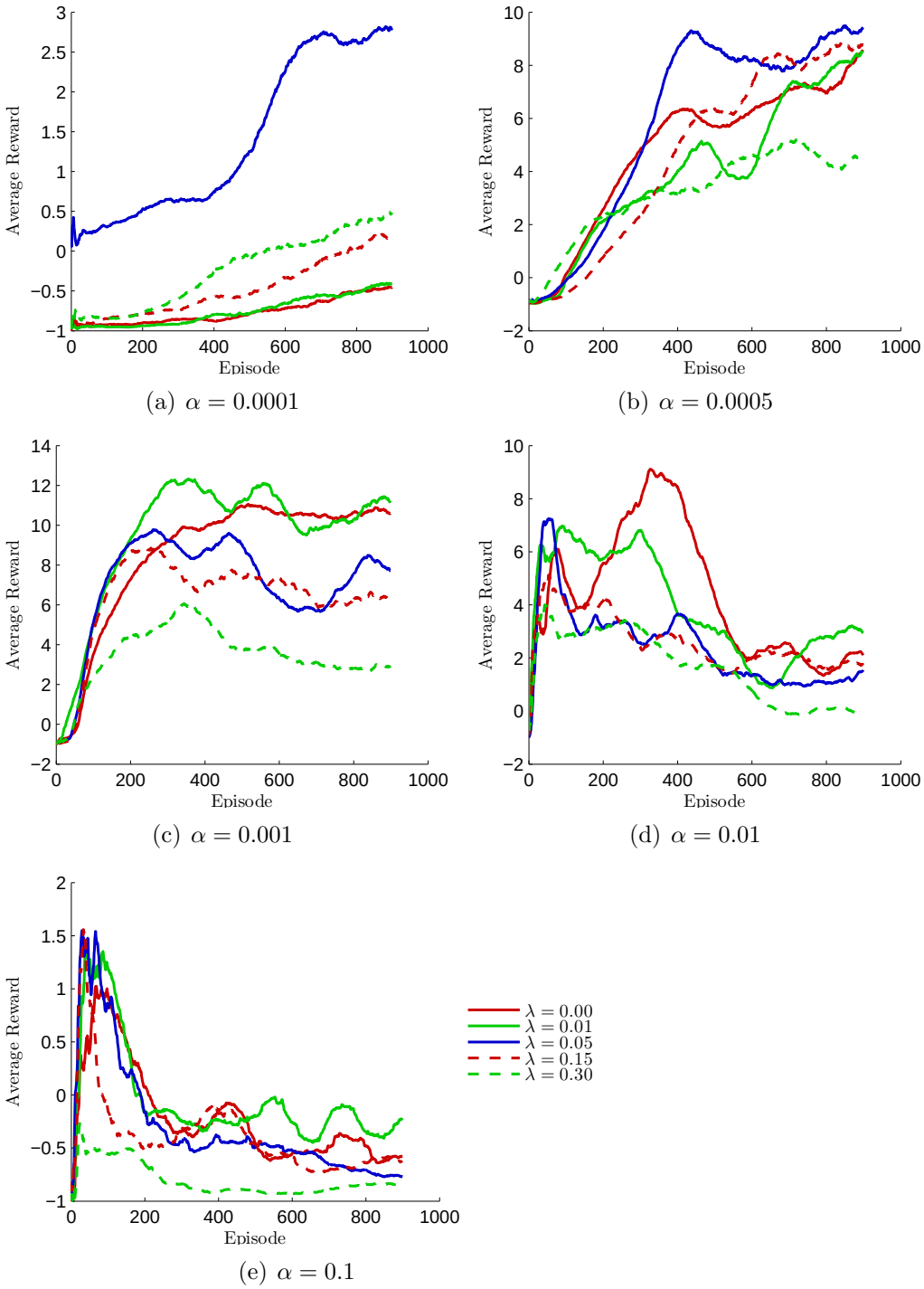


Figure 11. The effect of learning parameters on the average performance of the agent in accumulating rewards, over time, is depicted in each graph, grouped by α.

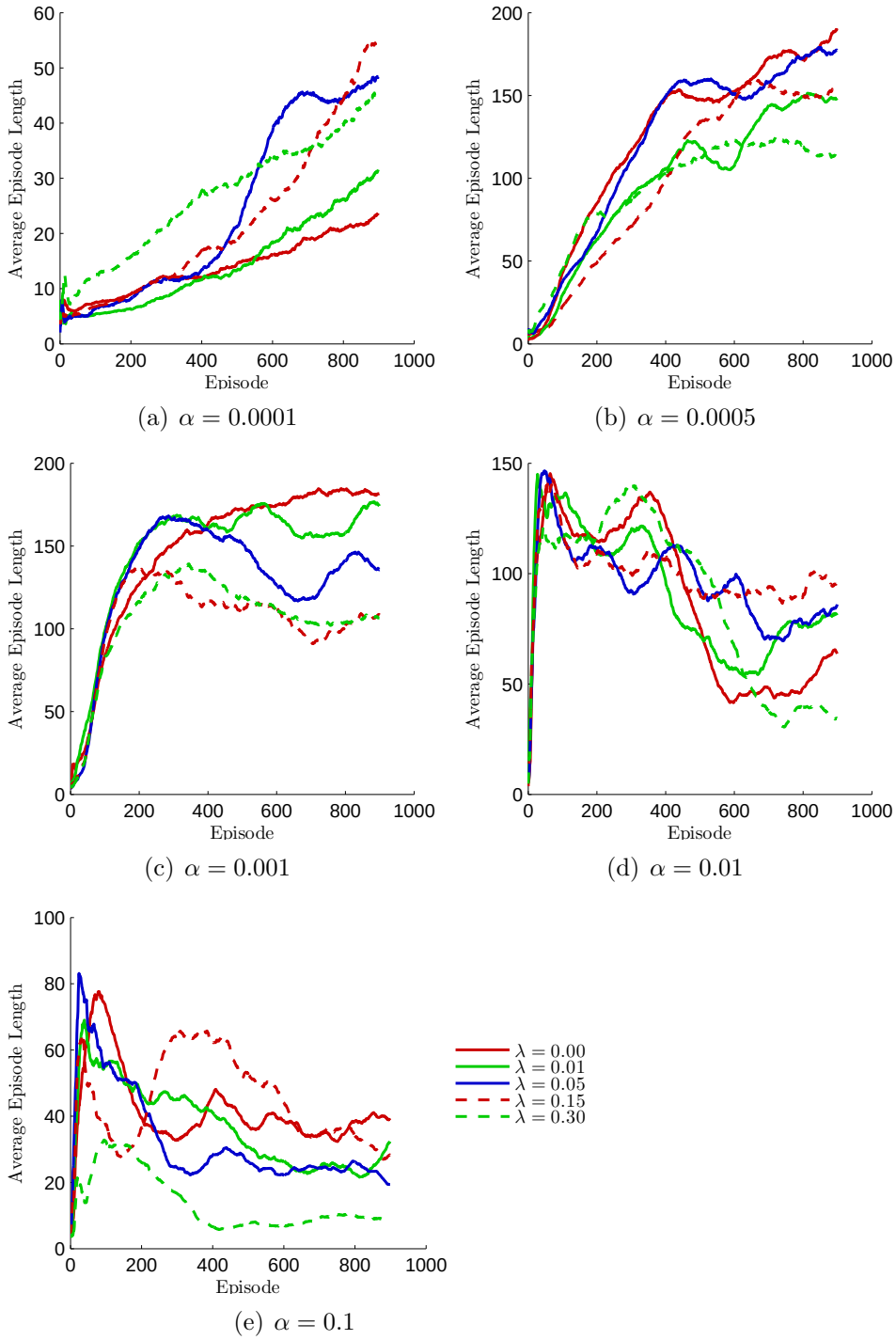


Figure 12. The effect of learning parameters on the agent's average length of game, over time, is depicted in each graph, grouped by α.

describe the agent's performance.

Stability

There was a great effect on learning stability at different parameter levels. When a learning curve trends upward and maintains convergence at a specific level, it is said to be stable. Often times, for bad choices of α and λ, agent performance would initially improve very quickly but would immediately degrade to lower values and never return. This was especially true for higher values of α.

As seen in the Random Walk example in Chapter II, increasing values of λ seemed to increase the variation and the noise present in performance trends. Though with $\lambda = 0.01$ having a slightly higher performance averages, observation of $\lambda = 0.00$ seemed to trend more predictably.

Convergence

For the stable configurations, the primary effect of α controlled the speed of convergence. Though it is shown for, high values of α, higher performance metrics were not maintained, as the agent's learning diverged. For $\alpha = 0.001$, the agent showed consistently higher and stable performance than at any other level. Lesser values took longer for the agent to reach peak performance; learning with $\alpha = 0.0001$ looked like it would take many times longer than that of $\alpha = 0.0001$ or $\alpha = 0.0005$.

Summary

The computer player was more capable of reaching a higher peak performance more quickly when $\alpha = 0.001$. The effect of λ generally decreased the agent's performance with increasing values. Though with $\lambda = 0.00$ being more stable, the trial averages and learning curves show that $\lambda = 0.01$ was more capable of trending to a higher level of performance than at any other level. Therefore, the

best configuration of these parameters for future trial sets will be configured such that $\lambda = 0.01$ and $\alpha = 0.001$.

2 Trial Set Two

With λ and α set to the best configuration identified by the previous trials, the agent's neural network structures were tested. For both the actor and critic, different numbers of hidden nodes and layers were tested for any noticeable response in the agent performance. In the following set of trials, the agent will be configured such that the actor and critic networks will have similar structure; they each will have the same number of hidden nodes and number of layers. This is due to the fact that preliminary trials show a great divergence when the two networks have different structures. The reasons for this are unknown (but can be speculated.) In the interest of seeking the best solution, this restriction is adopted.

Each trial sets the learning agent to a specific number of nodes and layers, with the performance recorded for 1000 games. Each trial was repeated 10 times. The average number of rewards per episode and the average episode length for each configuration is shown in Tables 1 and 2. The number of total layers for the actor anc critic is denoted, L as in Chapter II and the number of nodes per layer is presented here as J.

As in the previous set of trials, the average learning curves for the tested configurations are presented in two sets in Figures 13, 14, 15, and 16. Figures 13 and 14 show how the learning curves differ by variations in J with L held constant. Figures 15 and 16 allow direct comparisons of changes in L with J held constant.

Correlation of Reward and the Length of Game

Unlike the last set of trials where parameters specific to learning rates were tested, there are some factor configurations that illustrate stark differences between the two performance metrics. In this particular case of factors, the metrics

TABLE 3

Effect of neural network structure on the average reward accumulated per game. Each trial tested a different configuration of network parameters: the number of nodes per layer, J, and the number of layers, L

Average Reward Per Episode				
J	$L = 2$	$L = 3$	$L = 4$	$L = 5$
10	-0.74	-0.18	0.41	0.44
30	1.02	6.37	5.7	7.35
50	1.41	7.49	7.62	4.32
70	1.05	8.02	6.41	6.01
90	1.39	8.01	5.78	4.36

TABLE 4

Effect of neural network structure on the average length of game. Each trial tested a different configuration of network parameters: the number of nodes per layer, J, and the number of layers, L

Average Length of Episode				
J	$L = 2$	$L = 3$	$L = 4$	$L = 5$
10	12.32	18.97	32.61	28.91
30	62.98	108.16	111.89	127.76
50	74.86	127.91	154.74	149.77
70	96.38	142.65	152.41	151.71
90	100.34	159.65	156.67	152.26

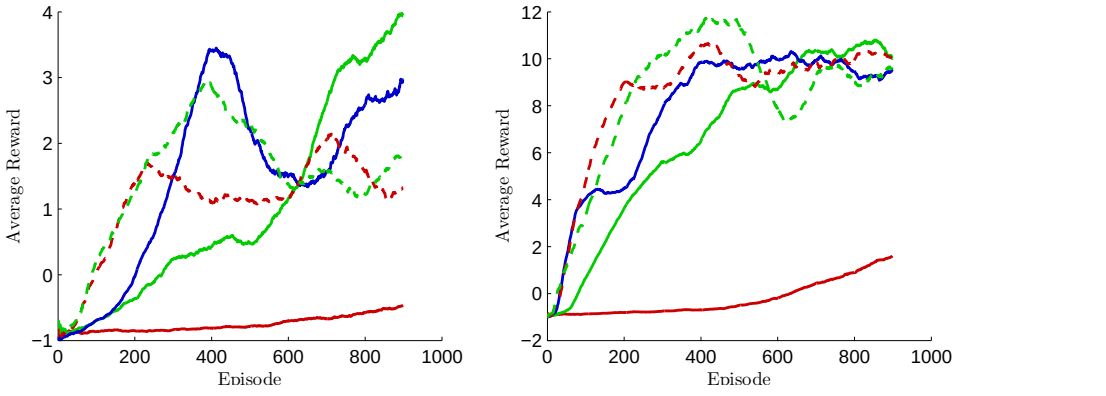


Figure 13. Effect of network structure on the accumulation of rewards, over time, is depicted in each graph, grouped by the number of network layers, L.

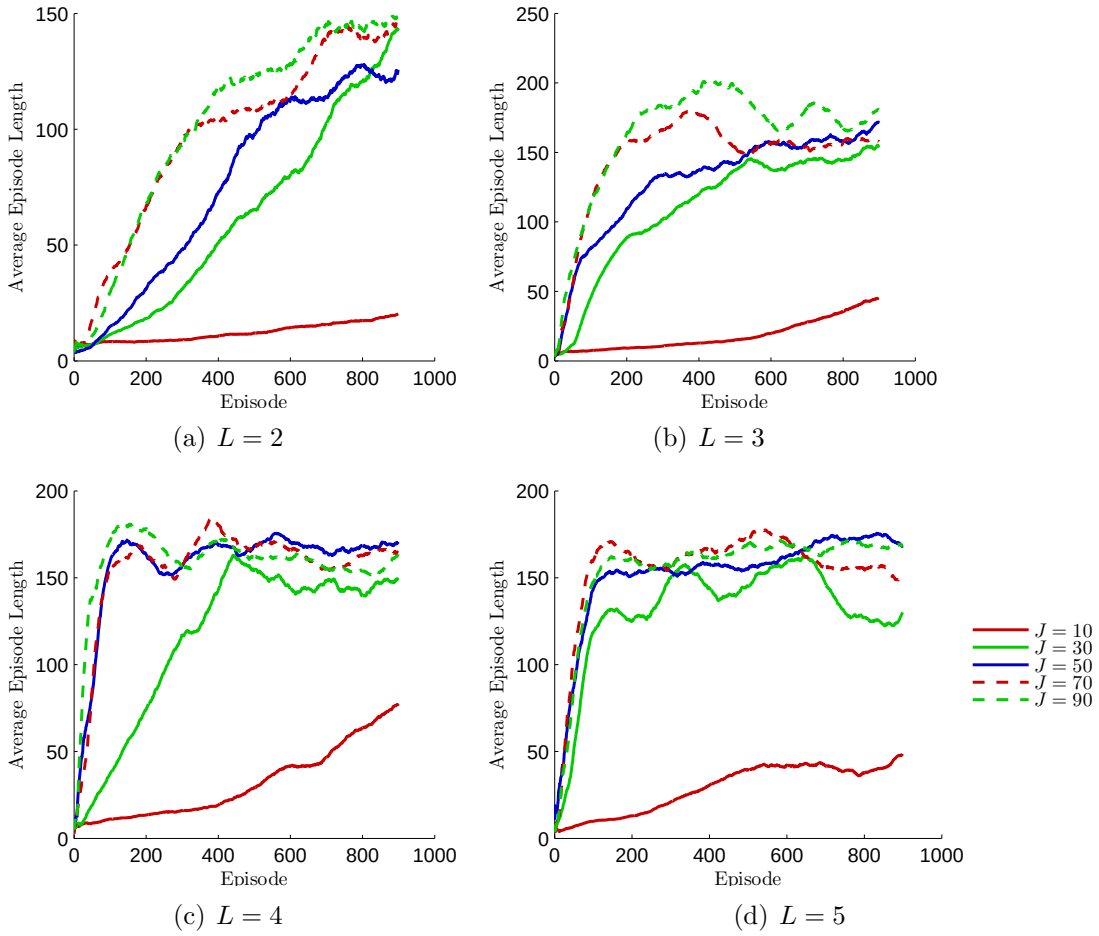


Figure 14. Effect of network structure on the agent's average length of game, over time, is depicted in each graph, grouped by the number of network layers L.

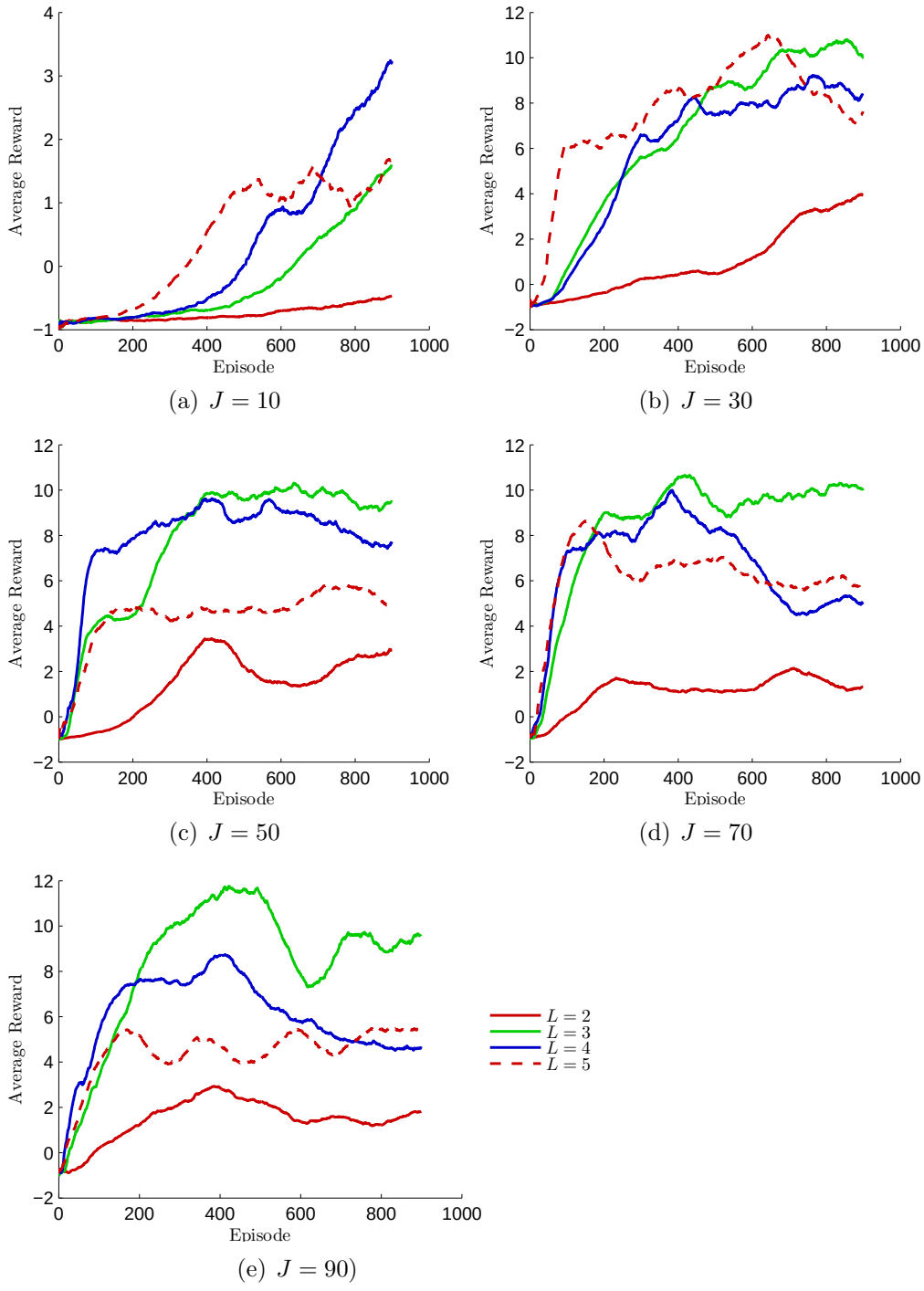
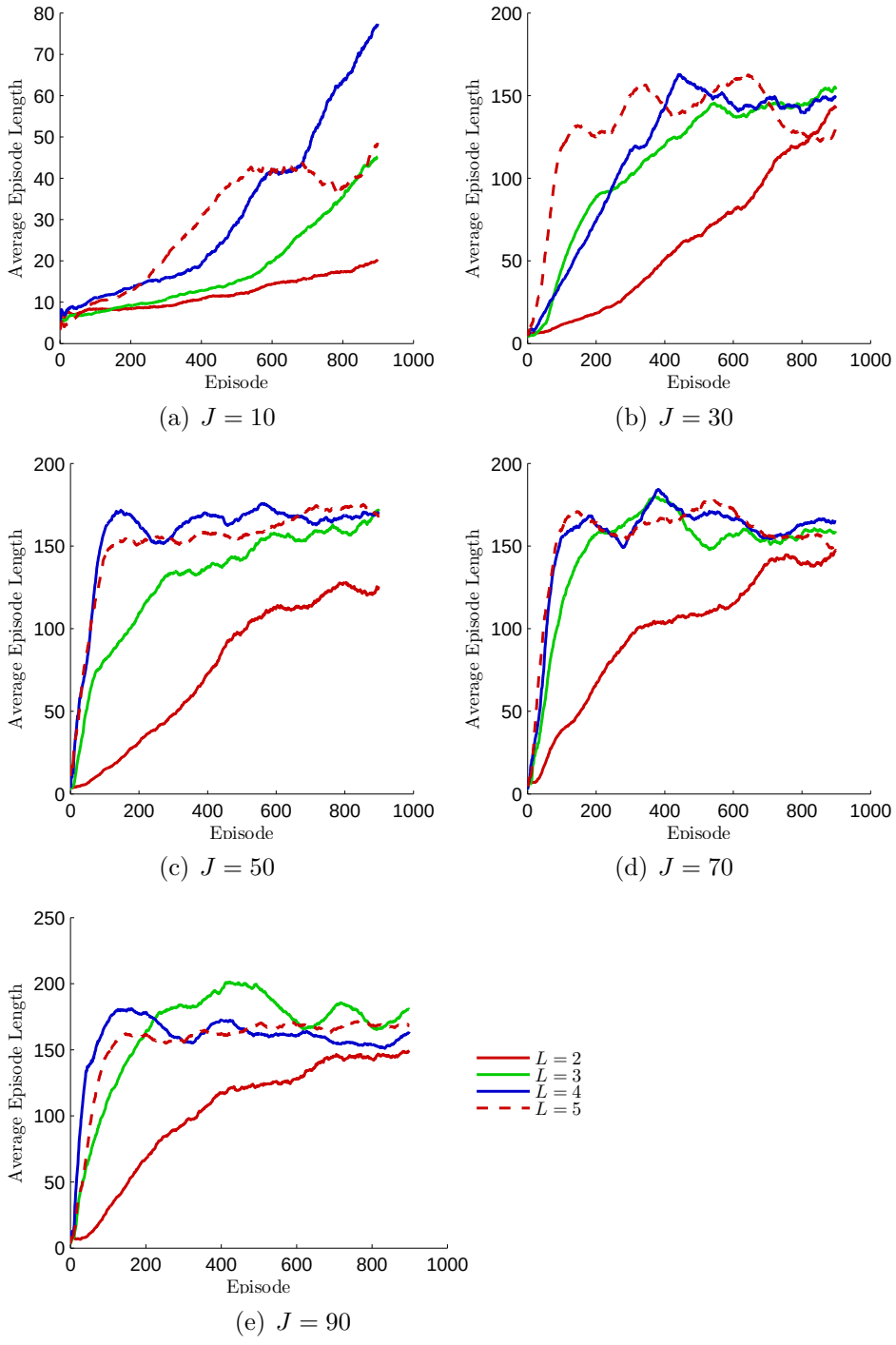


Figure 15. Effect of network structure on the accumulation of rewards, over time, is depicted in each graph, grouped by the number hidden nodes per layer, J.



(a) $J = 10$ (b) $J = 30$

(c) $J = 50$ (d) $J = 70$

(e) $J = 90$

Figure 16. Effect of network structure on the agent's average length of game, over time, is depicted in each graph, grouped by the number hidden nodes per layer, J.

individually cannot describe the performance of the agent fully; there is a detectable imbalance in the skills of the agent where factor levels contribute to a greater performance in the average length of game versus the number of rewards accumulated per game.

Primarily, the number of layers in the networks created the most significant unbalance between episode length and accumulated rewards. For the variations on the number of nodes per layer, J, the growth of the length of episodes was very consistent among all factor levels of L. For $J >= 30$, the agent very quickly learned to maximize the length of episode in all cases. For other levels of L, there was a great difference in accumulating rewards.

Convergence

The number of layers in a network had a significant effect on the convergence of the network. Increasing the number of nodes and layers did not guarantee better performance. In fact, choosing $L = 5$ restricted the peak performance of the network. On the other hand, not having enough layers slowed the learning growth. In terms of learning to collect rewards, $L = 3$ and $L = 4$ perform reasonably well, both converging quickly and to higher values.

Summary

In the tables with $L = 3$, $J = 70$ accumulated the total greatest reward which was only slightly better than $J = 50$. Since both choices perform well, and since the reward performance was generally more stable for $J = 50$, in following trials, agents will be constructed with 3 layers at 50 nodes per hidden layer.

3 Trial Set Three

This set of trials focuses on determining the best variation of state representations using the best configuration of parameters discovered thus far. The

different variations as proposed in the previous chapter are tested. It is suspected that states with more information may take longer to converge but can potentially perform better.

Five states representations will be tested, the two important feature states adopted from Ehlis' initial feature set, two sizes of the visual state, and a the state that is a combination of the two. Since it may take longer for the networks to learn to interpret more information (as the visual input is higher in dimensions) the length of the trial was extended to 2000 episodes in an attempt to observe peak performance. As before, each trial was repeated 10 times. The performance results are shown in the learning curves in Figures 3 and 18.

Summary

It is clear from the performance chart that the choice of visual state greatly affected the agent's ability to play the game. The hypothesis that the more environment information the agent had would translate into better play, was incorrect. The best state representation was the simplest. The adopted state vector adopted from Ehlis' initial feature set by far outperformed the rest, in rewards and length of game. The increase in performance observed by Ehlis using the improved second feature set, was not seen in this experiment. As the dimensionality increases, it would seem to result in lower agent performance.

It may be possible that the network parameters (such as the number of layers, nodes, and learn rate) favor the important feature state as that is what was used during the previous trials. It is not unreasonable to assume that a different feature set may entice a different network and leaning configuration. It is left for future work to determine what are the optimal configurations for the sets.

For the remainder of the experiment and for the technique comparison, Ehlis' initial feature set will be used.

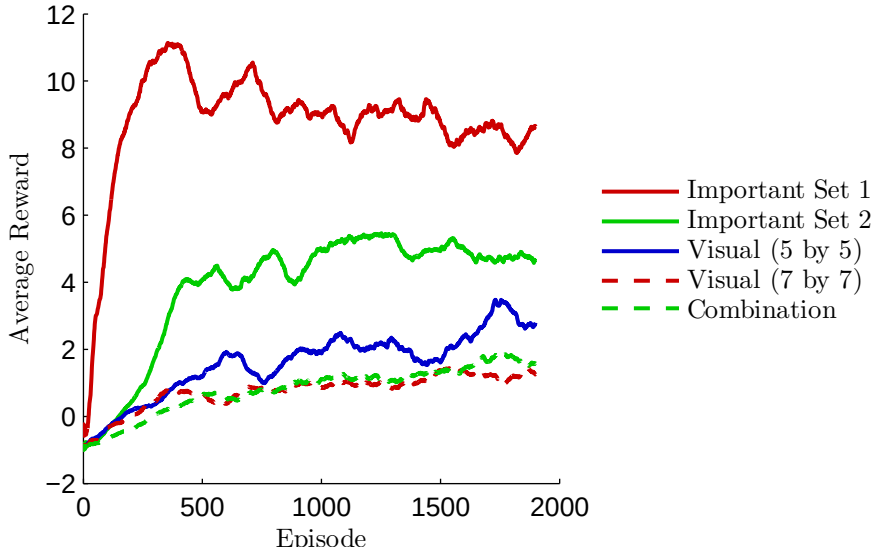


Figure 17. The effect of different state representations on the agent's ability to collect rewards.

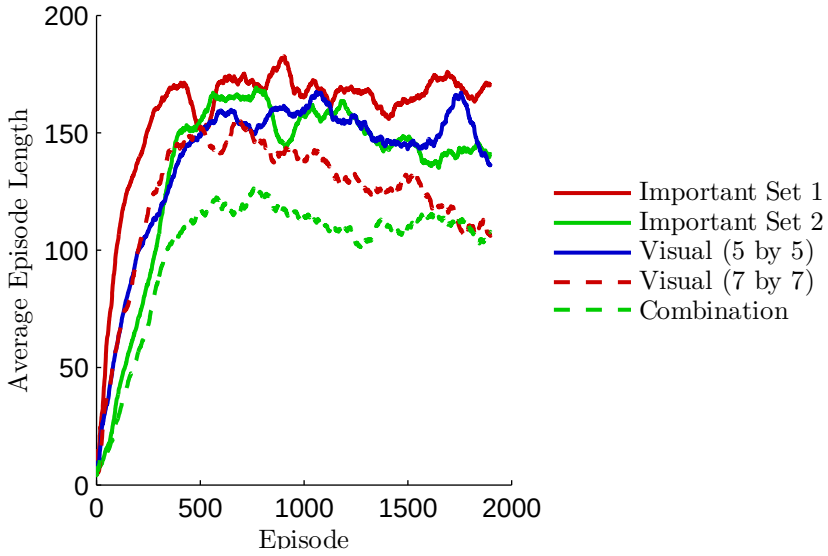


Figure 18. The effect of different state representations on the agent's length of game.

4 Trial Set Four

In this set, the SoftMax action selection temperature parameter τ is tested. The SoftMax temperature is responsible for controlling the probability of selecting actions. A higher value will make the agent more exploitative as action selections will approach equality. The lower it is, the more likely the most preferred action will be taken. The performance effect of changing the τ parameter is shown in Figures 4 and 20.

Summary

In terms of the average length of game, it is clear that most of the values perform equally. However, when $\tau = 10$ the agent rises to the same level of performance, it just takes longer for it to get there. The effect is similar in terms of the average rewards per game except there is a clearer difference between individual parameter curves. When $\tau = 1$, the agent achieved the greatest total rewards, as it appears that there is a maximum response near this point. The τ parameter will be set to 1 for the comparison study.

E Experiment Summary

The goal of the experiment was to find the best set of parameters that allowed an agent, controlled by the proposed TD learning technique, to learn to achieve the highest score possible. Using the two performance measures, as proposed in the previous chapter, different configurations of the learning system were tested in trial sets. The trial sets determined the best values of the modifiable parameters:

Learn rate, α: 0.001

Trace decay, λ: 0.01

Number of network layers, L: 3

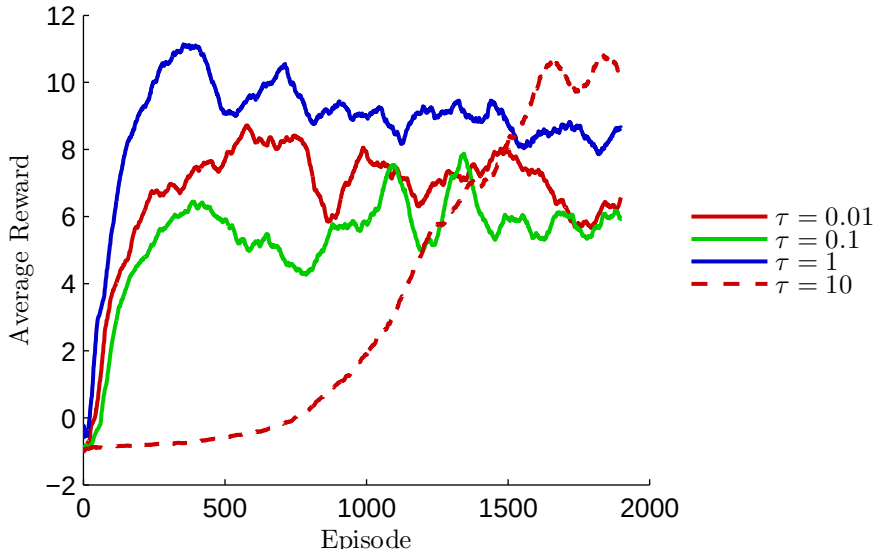


Figure 19. The effect of the SoftMax temperature parameter on the agent's ability to collect rewards.

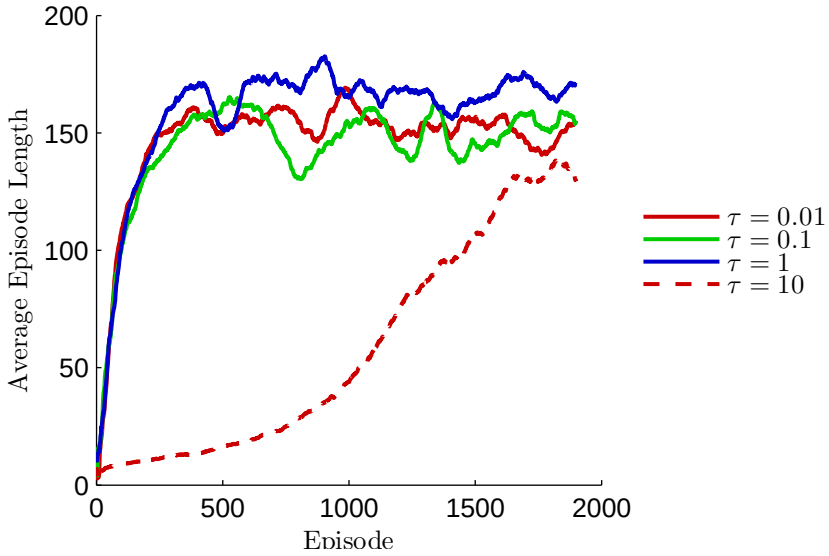


Figure 20. The effect of the SoftMax temperature parameter on the agent's length of game.

Number of nodes per hidden layer, J: 50

State vector presentation, $\vec{s}$: The set of important features

SoftMax action selection temperature, τ: 1

This configuration of parameters will be used to train an agent for a direct comparison to Ehlis' genetic programming solution.

CHAPTER V

METHOD COMPARISON AND EVALUATION

As stated, the best configuration of parameters and system factors will be used to evaluate the proposed temporal difference technique against an existing machine learning solution. In Chapter II, it was introduced that Ehlis was capable of creating an agent that could learn to play a perfect game of Snake. The genetic program that resulted from his work was published. Using the same performance metrics used in the previous experiment, a single trial will be conducted to compare the agents produced by both techniques.

A Developing Agents for Comparison Using Off-line Learning

The best TD computer agent must be generated from the experimentally determined configuration parameters. This occurs through a staged process. An off-line learning procedure is introduced to pre-train a set of initial agents. The best agent of the lot is selected through an evaluation trial and will be used to illustrate the differences between the machine learning techniques.

The agents will learn during an initial off-line training session. Training off-line means that the agents will reach their maximum performance before the evaluation trial is used to select the highest performer. This also implies that learning does not occur during evaluation; the agent will act with a consistent policy that will not change overtime.

Not all agents will train to the same level of performance. The differences are caused by the distribution of randomized weights; some initial configurations can

evolve into better solutions than others. Therefore, an initial pool of agents is created for a training session. As the agents train, they are closely monitored throughout each episode. If a change in the agent's reward performance is detected such that it is higher than any of the previous observations for the agent during the session, the agent and all of its network weights and parameters will be saved to disk. This will ensure that the peak policy will be preserved.

The relative performance of the agents in the pool will be evaluated. The highest overall performer will be selected for comparison with the genetic programming solution.

1 Results

Ten agents were pre-trained in an initial pool. The best performing agent was chosen for direct comparison to the solution produced by Ehlis. The ten agents were tested over a period of 500 episodes with the performance metrics recorded in Table 1 along with the statistics for the performance of the genetic programming agent, marked as GP.

In the table, the standard metrics are presented along with the *Steps per Reward*. It is an indicator of how quickly the agent retrieves food pieces; it is the average number of steps between successive rewards. The lower it is, the more quickly the agent collects the food.

Agents 6 and 8 managed to perform significantly better than the others, achieving the highest rewards in the 500 episode trial. Agent 9 was degenerate, and was incapable of learning any skill. Since agent 6 performed slightly better, it will be used in the comparison trial.

The performance of the best performing TD agent and the GP solution is presented in Figure 21 and Figure 22.

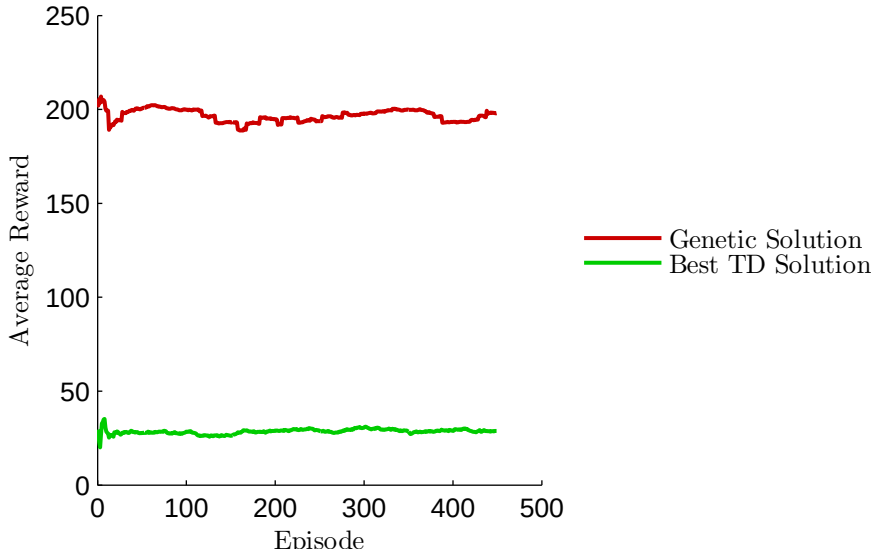


Figure 21. The average number of rewards per game recorded in a 500 episode trial testing both the best performing TD agent and the genetic programming (GP) solution by Ehlis.

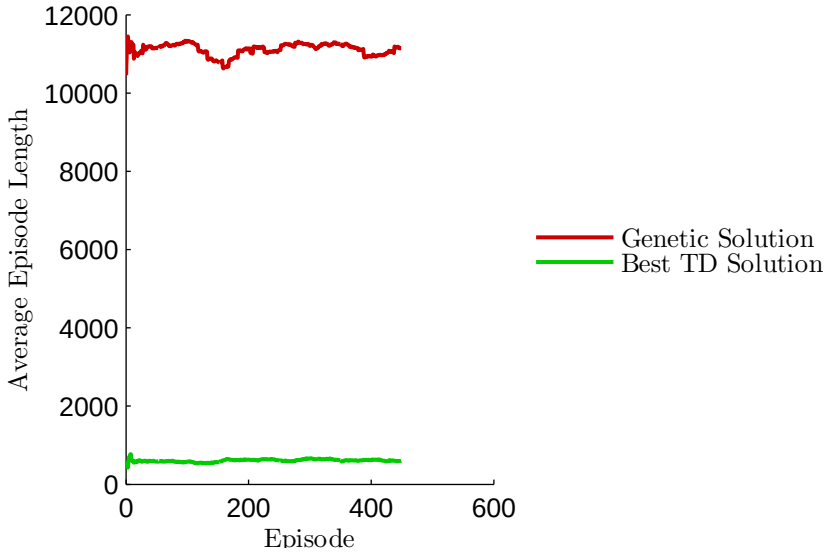


Figure 22. The average length of game recorded in a 500 episode trial testing both the best performing TD agent and the genetic programming (GP) solution by Ehlis. The GP solution is very inefficient.

TABLE 5

The statistics for the set of 10 initial agents trained with the TD learning technique introduced as compared to the Genetic Programming solution, GP.

Agent	Avg. Reward	Max Reward	Avg. Episode Length	Steps per Reward
1	1.66	15	164.92	45.45
2	19.8	38	491.14	24.39
3	23.44	42	500.65	20.83
4	20.34	44	419.57	20.00
5	22.13	49	484.57	16.67
6	28.53	58	607.76	20.83
7	19.45	39	485.25	25.64
8	27.88	56	595.67	20.83
9	-0.97	—	446.82	—
10	20.17	40	416.05	20.00
GP	197.60	211	11138.1	58.82

B Analysis of the proposed TD Method Against the Existing Technique

The GP solution comes very close to making a perfect score every episode it plays. These results show that the GP agent does not perform optimally all of the time. On average, it is less than 13 pieces of food per game from getting a perfect score every time, though it is quite capable of consuming the maximum 211 units of food.

The fact that the GP solution consistently plays very long lived games implies that it has excellent control but not a very good sense of the food. In fact, the way the GP agent achieves such high scores is that it almost entirely disregards the food piece in its movement strategy [2]. The agent follows a precise pattern on the playing field, zigzagging up and down while visiting every space causing the food to be eaten inevitably. For this reason. the GP agent exhibits the least efficient food gathering stategey. It takes an average of 58.8 steps for the GP agent to collect

one piece of food, while the agents in the TD group are rewarded 2–3 more frequently. Far from playing perfectly, the results shows that the TD solution can perform faster for the food it does collect.

Though it may be slow, gathering that many food bits and filling the entire field with Snake body is still impressive (though not very interesting to watch.) The behavior exhibited by the GP solution is very concrete, and predictable. Needless to say, it is not very human-like. The quality of this solution may not be desirable in a commercial game as it lacks the ability to replicate human realistic behavior.

1 TD Agent Behavior Analysis

Overall, the TD agents perform relatively well. While many humans can easily beat the scores exhibited and depending on the speed of the game, it is still a difficult task for a person. One subjective observation that was made is that many of the learned agents do appear to behave very human-like; the agents appear to move with intelligence as if someone of a medium skill level is playing the game. This could be seen a credit to this technique. It is possible that this TD method can create solutions that are *organic* in nature. If this is in-fact the case, a well performing solution using this configuration could potentially be used to create computer opponents in games, not unlike Tesauro's TD Gammon.

The most common strategy developed by the TD agents was the large circuit behavior that allowed the agent to travel between food pieces in the minimum number of steps. Agents that adopted this strategy, tended to strongly prefer turning in one direction to avoid danger. Each turn is calculated such that the food would be consumed on the next leg as it was detected. The turns were usually always made to left. This tendency can be identified as the main cause of terminating the game prematurely. When an agent is about to strike immediate danger, sometimes it is necessary that it avoids it in the opposite direction, in this case, to the right. If the tail is long enough, a turn in the same direction can cause

the agent to become trapped with the only option to spiral inward until inevitable doom. These situations are called *tail traps*.

It is believed that the tendency to always turn in the same direction is due to the fact that frequency of the negative rewards caused by the tail traps occurs much less frequently than the positive rewards received while circling. There may be a string of 20 to 30 rewards between a agent finds itself too long and spiraling inward to its demise. The numerous awards strengthens the preference for turning in the same direction when confronting danger; there are not near enough negative rewards to make the agent un-learn bad behaviors and learn new (and potentially beneficial) behaviors.

The assertion that the failure in the agents' inability to achieve a flawless performance (due to the imbalanced structuring of the rewards) assumes that there is a problem with the current reward policy. Since the reward policy defines the purpose and motivation for the agent to learn, it must be defined well so that the agent will learn equally well. It is proposed that the sub-optimal performance of the TD agents in this study can be attributed to one of the largest factors that were not specifically tested with different configurations, the reward policy.

2 Reward Policy and Expert Knowledge

The greedy learning behavior of the TD agents may be the cause of the agent's lack of high scores. The agents seek to maximize rewards, and like many eagerly selfish creatures, attempt to acquire the reward using the simplest and most direct technique. It is not unreasonable to consider the learning TD agents to be a lot like dogs: you can easily teach a dog a simple trick, like sitting or lying down by rewarding it as the action is happening. In the case of more complicated tricks, such as having Fido jump through a hoop of fire, a more complicated reward system would be needed to teach the appropriate strategy. By rewarding the dog in stages, giving treats for performing intermediate tasks, the animal can learn the needed

skills to perform the entire trick. Though more complicated, the learning result is a better performance than what could have been achieved through many direct attempts and painful failures.

The Snake playing TD agents potentially behave in the same way. In this study, they have learned to retrieve food in the simplest and most direct way. In order for the agent to collect food while keeping in a good position to avoid its tail, a reward policy that conveys good playing behaviors should be used. The problem is that the creation of such a policy requires more effort and would require expert knowledge of the environment; simply placing a reward when the desired task is complete is insufficient. Using expert knowledge in reinforcement learning is a drawback. Reinforcement techniques are primarily geared towards producing solutions without *a priori* knowledge. However, like most performance problems, there may be a trade-off. Future work should decide how much balance of expert knowledge is required to bring this technique up to the score level achieved by Ehlis' solution while at the same time, collecting rewards in an efficient manner.

A similar trade-off can be observed with the case of TD Gammon. When a feature set was constructed with expert knowledge such that the environment state represented carefully selected and compiled information, TD gammon was able to perform much better than with a plain representation of board positions. It was so successful using the second set of features that it was able to rise to a level of play worthy of competition with world-class players [10].

C Summary

Though the proposed system and system configurations did not learn to find an optimal performing solution to the game of Snake, there are a few merits of using the system identified. The agents produced by the system were able to play Snake moderately well: though the agents would gather food very quickly, the number of food pieces collected was limited by the agent's inability to learn strategy to avoid

falling into traps. It is suspected that such a system combined with more expert knowledge could potentially perform better. It is left as an exercise for future work.

CHAPTER VI

CONCLUSIONS AND RECOMENDATIONS

A Thesis Summary

The primary goal of this study was to evaluate the TDλ algorithm as applied to the game of Snake. The application of TD methods to this kind of problem results in many benefits: the study introduces a novel procedure for implementing artificial intelligence to the game of Snake while adding to the list of documented applications of reinforcement learning. The test of factors that affect the system's learning performance adds to the understanding of how the method responds to solving similar problems while establishing expectations for future results.

In the process of implementing a temporal difference solution to the game, there were several decisions regarding the selection the different components of the reinforcement system. The agent used function approximation with neural networks to estimate the value function for future reward. This required that an appropriate update procedure be defined for using multi-layer networks and the TD(λ) algorithm. In this piece of literature, the standard backpropagation algorithm used in traditional supervised learning was explicitly adapted to incorporate the temporal difference update procedure. For those who want to use a multilayer neural network with a TD system but, due to the scarcity of documentation in current literature, find the task daunting, this document may become a likely source for learning about the process.

Several forms of policy implementation could have been chosen fo use in the reinforcement learning system. In the proposed technique, actor-critic architecture

was used to learn a policy and define the action selection function. Using a multi-layer neural network, a process for updating actor eligibility traces was proposed and implemented for the actor. This undocumented procedure was explicitly defined such that others can benefit from the proposed technique.

The specific implementation considerations for the TD system, in relation to producing game playing agents, were fully documented. Performance concerns were identified as factors for an experiment that would attempt to find the configuration that maximized an agent's learning capabilities. The effects of the tested factors were clearly presented in learning curves and tables. The analysis of the performance response due to changes in the factors concluded a configuration that consistently achieved the highest level of play seen by the system. Though the agents were not capable of playing anywhere close to perfect games, they were able to reach a score comparable to that of a novice-intermediate player.

The TD solution to the game of Snake was compared directly to an existing machine learning solution. The results show that even though the previous solution was capable of achieving near perfect scores, the solutions found from the TD process was able to accumulate consume food at a faster rate but to a lesser magnitude. In addition, a subjective observation was made regarding the agent's ability to replicate human behaviors: the TD solution produced agents that behaved more life-like. Though capable of getting a perfect score, watching the genetic solution perform was found to be uninteresting and unappealing. The snake exhibited no sign of intelligence and followed a well defined pattern. The solution is not without merit, the machine intelligence is exhibited in the process of generating the solutions through the genetic programming algorithm. The behavioral observation of the genetic solution exemplifies the ability to exhibit human realistic behavior as an advantage to the proposed TD system.

The TD system was shown to be a viable way to implement simple artificial intelligence for simple games. It proved that it could create agents capable of

intelligent decisions. However, it also was shown that there are many factors which both affect and limit performance. When the complexity was increased by using visual data, the performance dropped dramatically. In modern digital games, where the state representation of the game could be quite large, the expected performance of this system would be inadequate. In addition, the on-line aspect of the unpredictable nature of learning is shown to be unstable. These qualities make it unlikely to be used in modern digital games. Until these limitations are surpassed and the unpredictable nature of the learning system becomes more consistent, the application of the system will remain targeted at simple games and academic exercises. The use of this TD system in creating computer controlled opponents capable of adapting complex behaviors and strategies does not appear to be possible with the current configuration.

B Future Work

The study proposed by this thesis attempted to answer many of questions about the implemented reinforcement learning system. However, the results obtained were far from perfect. Further research should be conducted to see what requirements exist to expand the system to achieve greater performance. Specifically, the implementation of expert data in a reward policy may lead to improved performance of the snake playing agent. The research should consider the not only the amount of expert knowledge used in crafting a reward policy, but also how many intermediate rewards and steps is necessary to teach an agent good player strategy, if possible.

In future trials, the ability of the agent to respond to different playing fields should be tested. The playing field used in this study was static and empty (except for the snake). Fields with pre-placed obstacles could provide interesting test cases in the evaluations of the agent's capabilities.

It has been suggested that the game may have a distinct set of phases. The

strategy at the beginning of the game, when the snake is short, could potentially be different near the end, where the snake is long. Training the snake for each identified phase of the game could bring interesting results. For example, if the snake were allowed to begin training with a body that fills half the board, how would the learned policy be different? This might allow the agent to learn to deal with the problems encountered in the middle phase. Further exploration into these considerations should be made. Potentially, a way may be found that further enhances the performance and merits of the system.

REFERENCES

[1] Wikimedia Commons. Artificial neural network, 2009. URL http://commons.wikimedia.org/w/index.php?title=Artificial_neural_network. [Online; accessed 9-April-2010].

[2] Tobin Ehlis. Application of genetic programming to the snake game. *GameDev.net: Genetic Algorithms*, 2000. URL http://www.gamedev.net/reference/articles/article1175.asp.

[3] Pablo Jose Funes. *Evolution of complexity in real-world domains*. PhD thesis, Waltham, MA, USA, 2001. Adviser-Pollack, Jordan B.

[4] Leo Galway, Darryl Charles, and Michaela Black. Machine learning in digital games: a survey. *The Artificial Intelligence Review*, 29(2):123–161, April 2009. URL http://www.proquest.com.echo.louisville.edu/.

[5] André LaMothe. *A Neural Net Primer*, pages 330–350. Charles River Media, 2000.

[6] Genevieve Orr, Nici Schraudolph, and Fred Cummins. Error backpropagaion, 1999. URL http://www.willamette.edu/ gorr/classes/cs449/backprop.html#top.

[7] Warren S.Sarle. Neural network faq, 2002. URL ftp://ftp.sas.com/pub/neural/FAQ.html.

[8] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 1988.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 1998.

[10] G. Tesauro. Temporal difference learning and td-gammon. *Commum ACM*, 38 (3):58–68, 1995.

CURRICULUM VITAE

NAME: Christopher Lockhart

ADDRESS: Department of Computer Engineering and Computer Science
University of Louisville
Louisville, KY 40292

EDUCATION: B.S. Computer Engineering and Computer Science
University of Louisville
2004

AWARDS: Gwong Sun Scholarship
CECS Department Alumni Award